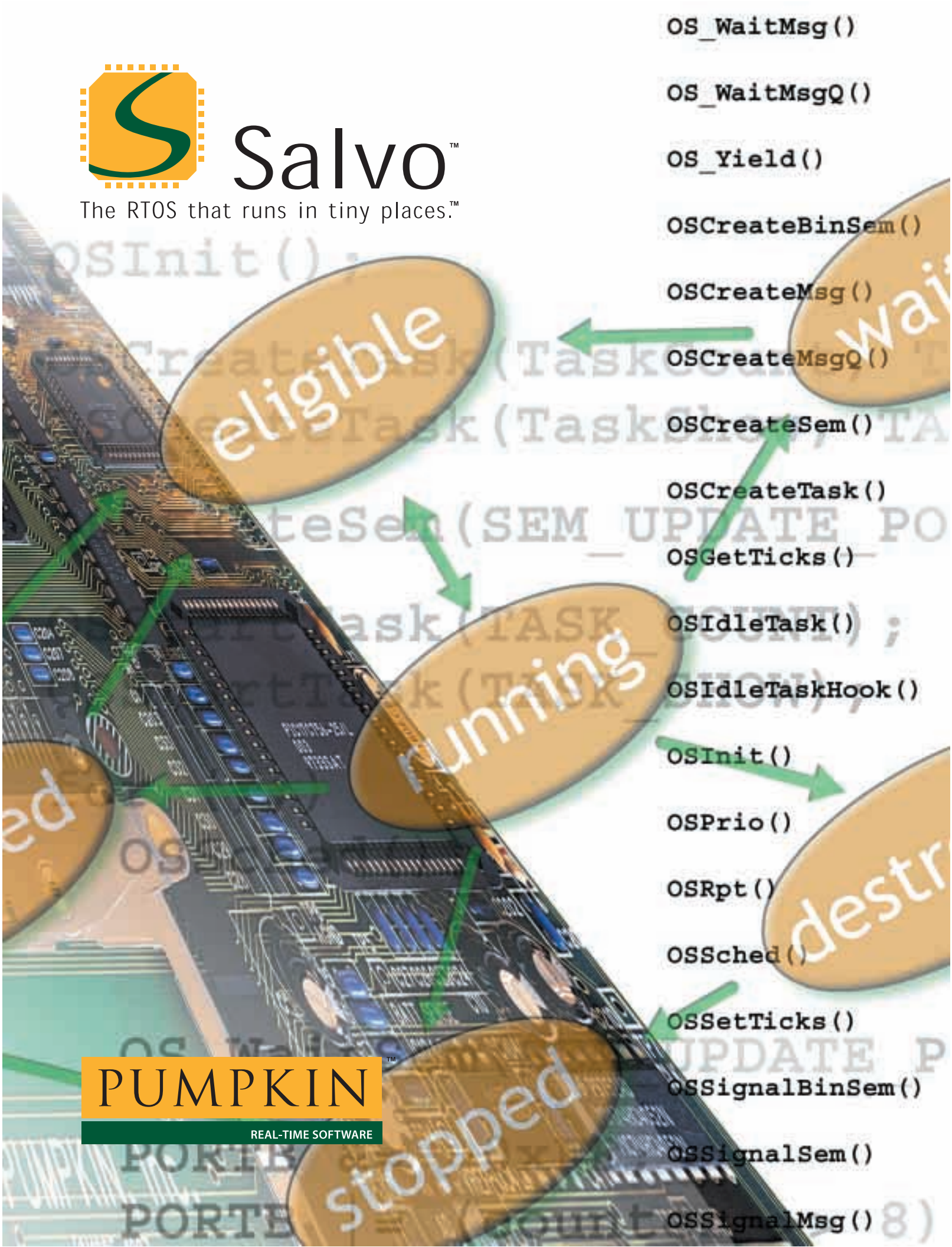

Salvo™
The RTOS that runs in tiny places.™
OS_WaitMsg()
OS_WaitMsgQ()
OS_Yield()
OSCreateBinSem()
OSCreateMsg()
OSCreateMsgQ()
OSCreateSem()
OSCreateTask()
OSGetTicks()
OSIdleTask()
OSIdleTaskHook()
OSInit()
OSPrio()
OSRpt()
OSSched()
OSSetTicks()
OSSignalBinSem()
OSSignalSem()
OSSignalMsg()
eligible
running
stopped
PUMPKIN™
REAL-TIME SOFTWARE

(inside front cover)



Salvo™

The RTOS that runs in tiny places.™

User Manual

version 3.1.0

for all distributions

PUMPKIN™

REAL-TIME SOFTWARE

Quick Start Guide

Thanks for purchasing Salvo, The RTOS that runs in tiny places.™ Pumpkin is dedicated to providing powerful, efficient and low-cost embedded programming solutions. We hope you'll like what we've made for you.

If this is the first time you've encountered Salvo, please review *Chapter 1 • Introduction* to get a flavor for what Salvo is, what it can do, and what other tools you'll need to use it successfully. See *Chapter 2 • RTOS Fundamentals* if you haven't used an RTOS before. Then try the steps below in the order listed.

Note You don't need to purchase Salvo to run the demo programs, try the tutorial or use the freeware libraries to build your own multitasking Salvo application – they're all part of Salvo Lite, the freeware version of Salvo.

Running a Demo

If you have a compatible target environment, you can run one of the standalone Salvo demonstration applications contained in `salvo\demo` on your own hardware. Open the demo's project, build it, download or program it into your hardware, and let it run. Most demo programs provide real-time feedback. If it's a Salvo Lite demo and uses commonly available hardware (e.g. `salvo\demo\d4`), you can even build your own application by modifying the source and re-building it.

See *Appendix C • File and Program Descriptions* for more information on the demo programs.

Trying the Tutorial

Chapter 4 • Tutorial builds a multitasking, event-driven Salvo application in six easy steps. The tutorial will familiarize you with Salvo's terminology, user services, and the process of building a working application. A set of tutorial projects is included with every Salvo distribution for embedded targets, enabling you to build each tutorial application by simply loading and building the project in the appropriate development environment.

Salvo Lite

A compiler that's certified for use with Salvo is all you need to use Salvo Lite, the freeware version of Salvo. You can write your own, small multitasking application with calls to Salvo services and link it to the freeware libraries. See *Chapter 4 • Tutorial* and the Salvo Application Note for your compiler and/or target for more information.

Even if you don't have a certified compiler, there may be a freeware version available – look in `salvo\free\links`.

Salvo LE

Salvo LE adds the standard Salvo libraries to Salvo Lite. This means that the numbers of tasks, events, etc. in your application are limited only by the available RAM.

Salvo Pro

With Salvo Pro, you'll have full access to all its source code, standard libraries, test programs and priority support. If you haven't done so already, try the tutorial in *Chapter 4 • Tutorial* as a first step towards creating your own application. Then use the configuration options in *Chapter 5 • Configuration* and the services outlined in *Chapter 7 • Reference*, along with their examples, to fine-tune Salvo to your application's requirements. If you run into problems or have questions, you'll find lots of useful information in *Chapter 6 • Frequently Asked Questions (FAQ)* and *Chapter 11 • Tips, Tricks and Troubleshooting*.

Getting Help

Some of the best resources for new and experienced Salvo users are the Salvo User Forums, hosted on Pumpkin's web site, http://www.pumpkininc.com/. Check there for up-to-date information on the latest Salvo releases.

Contact Information & Technical Support

Contacting Pumpkin

Pumpkin's mailing address and phone and fax numbers are:

Pumpkin, Inc.
750 Naples Street
San Francisco, CA 94112 USA
tel: 415-584-6360
fax: 415-585-7948

info@pumpkininc.com
sales@pumpkininc.com
support@pumpkininc.com

Time Zone: GMT–0800 (Pacific Standard Time)

Connecting to Pumpkin's Web Site

Use your web browser to access the Pumpkin web site at

http://www.pumpkininc.com/

Information available on the web site includes

- Latest News
- User Manuals
- Software Downloads & Upgrades
- Application Notes
- Assembly Guides
- Release Notes
- User Forums

Salvo User Forums

Pumpkin maintains User Forums for Salvo at Pumpkin's web site. The forums contain a wealth of practical information on using

Salvo, and is visited by Salvo users as well as Pumpkin technical support.

How to Contact Pumpkin for Support

Pumpkin provides online Salvo support via the Salvo Users Forums on the Pumpkin World Wide Web (WWW) site. Files and information are available to all Salvo users via the web site. To access the site, you'll need web access and a browser (e.g. Netscape, Opera, Internet Explorer).

Internet (WWW)

The Salvo User Forums are located at:

http://www.pumpkininc.com

and are the *preferred* method for you to post your pre-sales, general or technical support questions.

Email

Normally, we ask that you post your technical support questions to the Salvo User Forums on our website. We monitor the forums and answer technical support questions on-line.

In an emergency, you can reach technical support via email:

support@pumpkininc.com

We will make every effort to respond to your email requests for technical support within 1 working day. Please be sure to provide as much information about your problem as possible.

Mail, Phone & Fax

If you were unable to find an answer to your question in this manual, check the Pumpkin website and the Salvo user Forums (see below) for additional information that may have been recently posted. If you are still unable to resolve your questions, please contact us directly at the numbers above.

What To Provide when Requesting Support

Registered users requesting Salvo technical support should supply:

- The Salvo version number
- The compiler name and version number
- The user's source code snippet(s) in question
- The user's `salvocfg.h` file
- All other relevant files, details, etc.

Small code sections can be posted directly to the Salvo User Forums – see the on-line posting FAQ on how to use the UBB code tags (`[code]` and `[/code]`) to preserve the code's formatting and make it more legible.

If the need arises to send larger code sections, or even a complete, buildable project, please compress the files and email them directly to Salvo Technical support (see below). Please be sure to provide all necessary files to enable Technical Support to build your Salvo application locally in an attempt to solve your problem. Keep in mind that without the appropriate target system hardware, support in these cases is generally limited to non-runtime problem solving. Technical Support will keep all user code in strictest confidence.

Salvo User Manual

Copyright © 1995-2002 by Pumpkin, Inc.

All rights reserved worldwide. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior permission of Pumpkin, Inc.

Pumpkin, Inc.
750 Naples Street
San Francisco, CA 94112 USA

tel: 415-584-6360
fax: 415-585-7948
web: www.pumpkininc.com
email: sales@pumpkininc.com

Disclaimer

Pumpkin, Incorporated ("Pumpkin") has taken every precaution to provide complete and accurate information in this document. However, due to continuous efforts being made to improve and update the product(s), Pumpkin and its Licensor(s) shall not be liable for any technical or editorial errors or omissions contained in this document, or for any damage, direct or indirect, from discrepancies between the document and the product(s) it describes.

The information is provided on an as-is basis, is subject to change without notice and does not represent a commitment on the part of Pumpkin, Incorporated or its Licensor(s).

Trademarks

The Pumpkin name and logo, the Salvo name and logo, and "The RTOS that runs in tiny places." are trademarks of Pumpkin, Incorporated.

The absence of a product or service name or logo from this list does not constitute a waiver of Pumpkin's trademark or other intellectual property rights concerning that name or logo.

All other products and company names mentioned may be trademarks of their respective owners. All words and terms mentioned that are known to be trademarks or service marks have been appropriately capitalized. Pumpkin, Incorporated cannot attest to the accuracy of this information. Use of a term should not be regarded as affecting the validity of any trademark or service mark.

This list may be partial.

Patent Information

The software described in this document is manufactured under one or more of the following U.S. patents:

Patents Pending

Life Support Policy

Pumpkin, Incorporated's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Pumpkin, Incorporated. As used herein:

1) Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in

accordance with instructions for use provided in the labeling, can be reasonably expected to result in significant injury to the user.

2) A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

Refund Policy and Limited Warranty on Media

Pumpkin wants you to be happy with your Salvo purchase. That's why Pumpkin invites you to test drive Salvo before you buy. You can download and evaluate the fully functional Salvo freeware version Salvo Lite from the Salvo web site. If you have questions while you are using Salvo Lite, please don't hesitate to consult the Salvo User Forums, contact our support staff at support@pumpkininc.com, or contact Pumpkin directly.

Because of this free evaluation practice, and because the purchased version contains the complete source code for Salvo, Pumpkin does not offer refunds on software purchases.

Pumpkin will replace defective distribution media or manuals at no charge, provided you return the item to be replaced with proof of purchase to Pumpkin during the 90-day period after purchase. More details can be found in Section 11 Limited Warranty on Media of the Pumpkin Salvo License.

Documentation Creation Notes

This documentation was produced using Microsoft Word, Creative Softworx Capture Professional, CorelDRAW!, Adobe Photoshop, Adobe Illustrator and Adobe Acrobat.

Document name:	SalvoUserManual.doc (a Master document)
Template used:	User's Manual - Template (TT).dot
Last saved on:	21:10, Sunday, October 27, 2002
Total pages:	611
Total words:	112896

Credits

Author:	Andrew E. Kalman
Artwork:	Laura Macey, Elizabeth Peartree, Andrew E. Kalman
C-language Advice:	Russell K. Kadota, Clyde Smith-Stubbs, Dan Henry
Compiler Advice:	Matthew Luckman, Jeffrey O'Keefe

Pumpkin Salvo Software License Agreement v1.2

Please Read this Carefully and Completely Before Using this Software.

(Note: The Terms used herein are defined below in Section 1 Definitions)

Grant of License

This License Agreement is a legal agreement between You and Pumpkin, which owns the Software accompanied by this License or identified above or on the Product Identification Card accompanying this License or on the Product Identification Label attached to the product package. By clicking the Yes (i.e. Accept) button or by installing, copying, or otherwise using the Software or any Software Updates You agree to be bound by the terms of this License. If You do not agree to the terms of this License, Pumpkin is unwilling to license the Software to You, and You must not install, copy, or use the Software, including all Updates that You received as part of the Software. In such event, You should click the No (i.e. Decline) button and promptly contact Pumpkin for instructions on returning the entire unused Software and any accompanying product(s) for a refund. By installing, copying, or otherwise using an Update, You agree to be bound by the additional License terms that accompany such Update. If You do not agree to the terms of the additional License terms that accompany the Update, disregard the Update and the additional License terms that accompany the Update. In this event, Customer's rights to use the Software shall continue to be governed by the then-existing License.

1 Definitions

"License" means this document, a license agreement.

"You" means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License or a future version of this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares or beneficial ownership of such entity.

"Pumpkin" means Pumpkin, Incorporated and its Supplier(s).

"Original Code" means Source Code of computer software that is described in the Source Code Notice (below) as Original Code, and which, at the time of its release under this License is not already Covered Code governed by this License.

"Source Code" means the preferred form of the Covered Code for making modifications to it, including all modules it contains, plus any associated interface definition files, scripts used to control compilation and installation of an Executable, or a list of source code differential comparisons against either the Original Code or another well known, available Covered Code of Your choice.

"Covered Code" means the Original Code or Modifications or the combination of the Original Code and Modifications, in each case including portions thereof.

"Executable" means Covered Code in any form other than Source Code.

"Application" means computer software or firmware that is created in combination with Covered Code.

"Software" means the proprietary computer software system owned by Pumpkin that includes but is not limited to software components (including, but not limited to Covered Code), product documentation and associated media, sample files, extension files, tools, utilities and miscellaneous technical information, in whole or in part.

"Update" means any Software Update.

"Larger Work" means a work that combines Covered Code or portions thereof with code not governed by the terms of this License.

"Modifications" means any addition to or deletion from the substance or structure of either the Original Code or any previous Modifications. When Covered Code is released as a series of files, a Modification is (i) any addition to or deletion from the contents of a file containing Original Code or previous Modifications, or (ii) any new file that contains any part of the Original Code or Previous Modifications.

"Support" means customer support.

"Prerelease Code" means portions of the Software identified as prerelease code or "beta" versions.

2 Copyright

The Software, including all applicable rights to patents, copyrights, trademarks and trade secrets, is the sole and exclusive property of Pumpkin, Incorporated and its Licensor(s) and is provided for Your exclusive use for the purposes of this License. The Software is protected by United States copyright laws and international treaty provisions. Therefore, You must treat the Software like any other copyrighted material, except that You may either (i) make one copy of the Software in machine readable form solely for backup or archival purposes, or (ii) transfer the Software to a hard disk, provided You keep the original solely for backup and archival purposes. Additionally, only so long as the Software is installed only on the permanent memory of a single computer and that single computer is used by one user for at least 80% of the time the computer is in use, that same user may also make a copy of the Software to use on a portable or home computer which is primarily used by such user. As an express condition of this License, You must reproduce and include on each copy any copyright notice or other proprietary notice that is on the original copy of the Software supplied by Pumpkin. You may not copy the printed materials accompanying the Software.

3 Source Code License

3.1 The Software is licensed, not sold, to You by Pumpkin for use only under the terms of this License, and Pumpkin reserves any rights not expressly granted to You. Except where explicitly identified as such, the Software is neither "shareware" nor "freeware" nor "communityware." The Software contains intellectual property in the form of Source Code, algorithms and other manifestations. You own the media on which the Software is recorded or fixed, but Pumpkin, Incorporated and its Licensor(s) retains ownership of the Software, related documentation and fonts.

3.2 Pumpkin grants You the use of the Software only if You have registered the Software with Pumpkin by returning the registration card or by other means specified by Pumpkin.

3.3 Pumpkin grants You a non-exclusive, worldwide License, subject to third-party intellectual property claims, (i) to use and modify ("Utilize") the Software (or portions thereof) with or without Modifications, or as part of a Larger Work, on a single computer for the purpose of creating, modifying, running, debugging and testing Your own Application and any of its updates, enhancements and successors, and (ii) under patents now or hereafter owned or controlled by Pumpkin, to Utilize the Software (or portions thereof), but solely to the extent that any such patent is reasonably necessary to enable You to Utilize the Software (or portions thereof) and not to any greater extent that may be necessary to Utilize further Modifications or combinations. To use ("Use") the Software means that the Software is either loaded in the temporary memory (i.e. RAM) of a computer or installed on the permanent memory of a computer (i.e. hard disk, etc.). You may Use the Software on a network, provided that a licensed copy of the software has been acquired for each person permitted to access the Software through the network. You may also Use the Software in object form only (i.e. as an Executable) on a single, different computer or computing device (e.g. target microcontroller or microprocessor, demonstration or evaluation board, in-circuit emulator, test system, prototype, etc.).

3.4 Any supplemental software code or other materials provided to You as part of Pumpkin's Support shall be considered part of the Software and subject to the terms and conditions of this License. With respect to technical information You provide to Pumpkin as part of the Support, Pumpkin may use such information for its business purposes, including product support and development. Pumpkin will not utilize such technical information in a form that personally identifies You without Your permission.

3.5 The Software shall be deemed accepted by You upon payment of the Software by You and shall not be granted a refund of any license fees for the Software, except for Your rights defined in this License.

4 Software Distribution Obligations

4.1 You may not under any circumstances release or distribute the Source Code, with or without Modifications, or as part of a Larger Work, without Pumpkin's express written permission.

4.2 You may distribute the Software in Executable form only and as part of a Larger Work only (i.e. in conjunction with and as part of Your Application. Additionally, You must (i) not permit the further redistribution of the Software in any form by Your customers, (ii) include a valid copyright notice in Your application (where possible - if it is not possible to put such a notice in Your Application due to its structure, then You must include such a notice in a location (such as a relevant directory file) where a user would be likely to look for such a notice), (iii) include the existing copyright notice(s) in all Pumpkin Software used in Your Application, (iv) agree to indemnify, hold harmless and defend Pumpkin from and against any and all claims and lawsuits, including attorney's fees, that arise or result from the use or distribution of Your Application, (v) otherwise comply with the terms of this License, and (vi) agree that Pumpkin reserves all rights not expressly granted.

4.3 You may freely distribute the demonstration programs (identified as "Demo") that are part of the Software as long as they are accompanied by this License.

4.4 The freeware version (consisting of pre-compiled libraries, a limited number of source code files, and various other files and documentation) and identified as "Freeware" is governed by this license, with the following exceptions: The sole exception shall be for a Larger Work created exclusively with the freeware libraries that are part of the Software; in this case Pumpkin automatically grants You the right to distribute Your Application freely.

4.5 You may not under any circumstances, other than those explicitly mentioned in Sections 4.2, 4.3 and 4.4 above, release or distribute the Covered Code, with or without Modifications, or as part of a Larger Work, without Pumpkin's express written permission.

5 Other Restrictions

5.1 You may not permit other individuals to use the Software except under the terms of this License.

5.2 You may not rent, lease, grant a security interest in, loan or sublicense the Software; nor may You create derivative works based upon the Software in whole or in part.

5.3 You may not translate, decompile, reverse engineer, disassemble (except and solely to the extent an applicable statute expressly and specifically prohibits such restrictions), or otherwise attempt to create a human-readable version of any parts of the Software supplied exclusively in binary form.

5.4 If the Software was licensed to You for academic use, You may not use the software for commercial product development.

5.5 You may not remove any designation mark from any supplied material that identifies such material as belonging to or developed by Pumpkin.

5.6 You may permanently transfer all of Your rights under this License, provided You retain no copies, You transfer all of the Software (including all component parts, the media and printed materials, any upgrades, and this License), You provide Pumpkin notice of Your name, company, and address and the name, company, and address of the person to whom You are transferring the rights granted herein, and the recipient agrees to the terms of this License and pays to Pumpkin a transfer fee in an amount to be determined by Pumpkin and in effect at the time in question. If the Software is an upgrade, any transfer must include all prior versions of the Software. If the Software is received as part of a subscription, any transfer must include all prior deliverables of Software and all other subscription deliverables. Upon such transfer, Your License under this Agreement is automatically terminated.

5.7 You may use or transfer the Updates to the Software only in conjunction with Your then-existing Software. The Software and all Updates are licensed as a single product and the Updates may not be separated from the Software for use at any time.

6 Termination

This License is effective until terminated. This License will terminate immediately without notice from Pumpkin or judicial resolution if You fail to comply with any provision of this License, and You may terminate this License at any time. Upon such termination You must destroy the Software, all accompanying written materials and all copies thereof. Provisions which, by their nature, must remain in effect beyond the termination of this License shall survive.

7 Multiple Media

Even if this Pumpkin product includes the Software on more than one medium (e.g., on both a CD-ROM and on magnetic disk(s); or on both 3.5 inch disk(s) and 5.25 inch disk(s)), You are only licensed to use one copy of the Software as described in Section 2.3. The restrictions contained herein apply equally to hybrid media that may contain multiple versions of the Software for use on different operating systems. Regardless of the type of media You receive, You may only use the portion appropriate for Your single user computer / workstation. You may not use the Software stored on the other medium on another computer or common storage device, nor may You rent, lease, loan or transfer it to another user except as part of a transfer pursuant to Section 5.7.

8 Prerelease Code

Prerelease Code may not be at the level of performance and compatibility of the final, generally available product offering, and may not operate correctly and may be substantially modified prior to first commercial shipment. Pumpkin is not obligated to make this or any later version of the Prerelease Code commercially available. The grant of license to use Prerelease Code expires upon availability of a commercial release of the Prerelease Code from Pumpkin.

9 Export Law Assurances

You may not use or otherwise export or re-export the Software except as authorized by United States law and the laws of the jurisdiction in which the Software was obtained. In particular, but without limitation, the Software may not be exported or re-exported to (i) into (or to a national or resident of) any U.S. embargoed country or (ii) to anyone on the U.S. Treasury Department's list of Specially Designated Nations or the U.S. Department of Commerce's Table of Denial Orders. By using the Software You represent and warrant that You are not located in, under control of, or a national or resident of any such country or on any such list.

10 U.S. Government End Users

If You are acquiring the Software and fonts on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees that the Software and fonts shall be classified as "commercial computer software" and "commercial computer software documentation" as such terms are defined in the applicable provisions of the Federal Acquisition Regulation ("FAR") and supplements thereto, including the Department of Defense ("DoD") FAR Supplement ("DFARS"). If the Software and fonts are supplied for use by DoD, it is delivered subject to the terms of this Agreement and either (i) in accordance with DFARS 227.7202-1(a) and 227.7202-3(a), or (ii) with restricted rights in accordance with DFARS 252.227-7013(c)(1)(ii) (OCT 1988), as applicable. If the Software and fonts are supplied for use by any other Federal agency, it is restricted computer software delivered subject to the terms of this Agreement and (i) FAR 12.212(a); (ii) FAR 52.227-19; or (iii) FAR 52.227-14(ALT III), as applicable.

11 Limited Warranty on Media

Pumpkin warrants for a period of ninety (90) days from Your date of purchase (as evidenced by a copy of Your receipt) that the media provided by Pumpkin, if any, on which the Software is recorded will be free from defects in materials and workmanship under normal use. Pumpkin will have no responsibility to replace media damaged by accident, abuse or misapplication. PUMPKIN'S ENTIRE LIABILITY AND YOUR SOLE AND EXCLUSIVE REMEDY WILL BE, AT PUMPKIN'S OPTION, REPLACEMENT OF THE MEDIA, REFUND OF THE PURCHASE PRICE OR REPAIR OR REPLACEMENT OF THE SOFTWARE. ANY IMPLIED WARRANTIES ON THE MEDIA, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY BY JURISDICTION.

12 Disclaimer of Warranty

THIS LIMITED WARRANTY IS THE ONLY WARRANTY PROVIDED BY PUMPKIN. PUMPKIN EXPRESSLY DISCLAIMS ALL OTHER WARRANTIES AND/OR CONDITIONS, ORAL OR WRITTEN, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE WITH REGARD TO THE SOFTWARE AND ACCOMPANYING WRITTEN MATERIALS, AND NONINFRINGEMENT. PUMPKIN DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE WILL BE CORRECTED. FURTHERMORE, PUMPKIN DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. AS A RESULT, THE SOFTWARE IS LICENSED "AS-IS", AND YOU THE LICENSEE EXPRESSLY ASSUME ALL LIABILITIES AND RISKS, FOR USE OR OPERATION OF ANY APPLICATION PROGRAMS YOU MAY CREATE WITH THE SOFTWARE, INCLUDING WITHOUT LIMITATION, APPLICATIONS DESIGNED OR INTENDED FOR MISSION CRITICAL APPLICATIONS AND HIGH-RISK ACTIVITIES, SUCH AS THE OPERATION OF NUCLEAR FACILITIES, PACEMAKERS, DIRECT LIFE SUPPORT MACHINES, WEAPONRY, AIR TRAFFIC CONTROL, AIRCRAFT NAVIGATION OR COMMUNICATIONS SYSTEMS, FACTORY CONTROL SYSTEMS, ETC., IN WHICH THE FAILURE OF THE SOFTWARE COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE. NO PUMPKIN DEALER, DIRECTOR, OFFICER, EMPLOYEE OR AGENT IS AUTHORIZED TO MAKE ANY MODIFICATION, EXTENSION, OR ADDITION TO THIS WARRANTY. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY BY JURISDICTION.

13 Limitation of Liabilities, Remedies and Damages

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL PUMPKIN, INCORPORATED, OR ANY OF ITS LICENSORS, SUPPLIERS, DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS (COLLECTIVELY "PUMPKIN AND ITS SUPPLIER(S)") BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL, INDIRECT OR SPECIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION AND THE LIKE, OR ANY OTHER PECUNIARY LOSS), WHETHER FORESEEABLE OR UNFORESEEABLE, ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE OR ACCOMPANYING WRITTEN MATERIALS, REGARDLESS OF THE BASIS OF THE CLAIM AND EVEN IF PUMPKIN AND ITS SUPPLIER(S) HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION WILL NOT APPLY IN CASE OF PERSONAL INJURY ONLY WHERE AND TO THE EXTENT THAT APPLICABLE LAW REQUIRES SUCH LIABILITY. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. IN NO EVENT SHALL PUMPKIN AND ITS SUPPLIER(S)' TOTAL LIABILITY TO YOU FOR ALL

DAMAGES, LOSSES AND CAUSES OF ACTION (WHETHER IN CONTRACT, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR OTHERWISE) EXCEED $50.00.

PUMPKIN SHALL BE RELIEVED OF ANY AND ALL OBLIGATIONS WITH RESPECT TO THIS SECTION FOR ANY PORTIONS OF THE SOFTWARE THAT ARE REVISED, CHANGED, MODIFIED, OR MAINTAINED BY ANYONE OTHER THAN PUMPKIN.

14 Complete Agreement, Controlling Law and Severability

This License constitutes the entire agreement between You and Pumpkin with respect to the use of the Software, the related documentation and fonts, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Pumpkin. The acceptance of any purchase order placed by You is expressly made conditional on Your assent to the terms set forth herein, and not those in Your purchase order. This License will be construed under the laws of the State of California, except for that body of law dealing with conflicts of law. If any provision of this License shall be held by a court of competent jurisdiction to be contrary to law, that provision will be enforced to the maximum extent permissible, and the remaining provisions of this License will remain in full force and effect. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any law or regulation that provides that the language of a contract shall be construed against the drafter shall not apply to this License. In the event of any action to enforce this Agreement, the prevailing party shall be entitled to recover from the other its court costs and reasonable attorneys' fees, including costs and fees on appeal.

15 Additional Terms

Nothing in this License shall be interpreted to prohibit Pumpkin from licensing under terms different from this License any code which Pumpkin otherwise would have a right to License.

This License does not grant You any rights to use the trademarks or logos that are the property of Pumpkin, Inc., even if such marks are included in the Software. You may contact Pumpkin for permission to display the above-mentioned marks.

Pumpkin may publish revised and/or new versions of this License from time to time. Each version will be given a distinguishing version number.

Should You have any questions or comments concerning this License, please do not hesitate to write to Pumpkin, Inc., 750 Naples Street, San Francisco, CA 94112 USA, Attn: Warranty Information. You may also send email to support@pumpkininc.com.

Source Code Notice

The contents of this file are subject to the Pumpkin Salvo License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.pumpkininc.com, or from warranty@pumpkininc.com.

Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for specific language governing the warranty and the rights and limitations under the License.

The Original Code is Salvo - The RTOS that runs in tiny places(tm). Copyright (C) 1995-2002 Pumpkin, Inc. and its Licensor(s). All Rights Reserved.

Contents

Contents i

Figures xix

Listings xxi

Tables xxiii

Release Notes xxvii

Introduction xxvii
Third-Party Tool Versions xxvii

Supported Targets and Compilers xxix

Preface xxxi

Typographic Conventions xxxi
Standardized Numbering Scheme xxxii
The Salvo Coding Mindset xxxiii
Configurability Is King xxxiii
Conserve Precious Resources xxxiii
Learn to Love the Preprocessor xxxiii
Document, But Don't Duplicate xxxiv
We're Not Perfect xxxiv

Chapter 1 • Introduction 1

Welcome 1
What Is Salvo? 2
Why Should I Use Salvo? 2
What Kind of RTOS Is Salvo? 3
What Does a Salvo Program Look Like? 3
What Resources Does Salvo Require? 5
How Is Salvo Different? 6
What Do I Need to Use Salvo? 7
Which compilers and Processors does Salvo support? 8
How Is Salvo Distributed? 8
What Is in this Manual? 8

Chapter 2 • RTOS Fundamentals 11

Introduction 11
Basic Terms 12
Foreground / Background Systems 14
Reentrancy 15
Resources 16
Multitasking and Context Switching 16
Tasks and Interrupts 17
Preemptive vs. Cooperative Scheduling 18
Preemptive Scheduling 19
Cooperative Scheduling 20
More on Multitasking 21
Task Structure 21
Simple Multitasking 22
Priority-based Multitasking 22
Task States 23
Delays and the Timer 24
Event-driven Multitasking 26
Events and Intertask Communications 29
Semaphores 29
Event Flags 29
Task Synchronization 31
Resources 33
Messages 35
Message Queues 37
Summary of Task and Event Interaction 37
Conflicts 38
Deadlock 38
Priority Inversions 39
RTOS Performance 39
A Real-World Example 39
The Conventional Superloop Approach 40
The Event-Driven RTOS Approach 41
Step By Step 43
Initializing the Operating System 43
Structuring the Tasks 43
Prioritizing the Tasks 44
Interfacing with Events 44
Adding the System Timer 45
Starting the Tasks 45
Enabling Multitasking 46
Putting It All Together 46
The RTOS Difference 48

Chapter 3 • Installation 51

Introduction 51
CD-ROM Installation 51
Internet Installation 56
Network Installation 57

Installing Salvo on non-Wintel Platforms 57
Installing Salvo Lite 57
A Completed Installation 58
Uninstalling Salvo 58
Uninstalling Salvo on non-Wintel Machines 59

Chapter 4 • Tutorial 61

Introduction 61
Part 1: Writing a Salvo Application 61
Initializing Salvo and Starting to Multitask 61
Creating, Starting and Switching tasks 63
Adding Functionality to Tasks 66
Using Events for Better Performance 68
Delaying a Task 72
Signaling from Multiple Tasks 76
Wrapping Up 80
Food For Thought 80
Part 2: Compiling a Salvo Application 80
Working Environment 80
Creating a Project Directory 81
Including salvo.h 82
Configuring your Compiler 82
Setting Search Paths 82
Using Libraries vs. Using Source Files 83
Using Libraries 83
Linking to Salvo Libraries 84
Building the Tutorial Program 86
Using Source Files 86
Setting Configuration Options 86
Including Salvo Files 90
Linking to Salvo Object Files 90
Building the Tutorial Program 91

Chapter 5 • Configuration 109

Introduction 109
Overview 109
Organization 110
Choosing the Right Options for your Application 112
Creating and Editing the Configuration File 114
Predefined Configuration Constants 114
Configuration Options for all Distributions 115
OSCOMPILER: Identify Compiler in Use 116
OSEVENTS: Set Maximum Number of Events 118
OSEVENT_FLAGS: Set Maximum Number of Event Flags 119
OSLIBRARY_CONFIG: Specify Precompiled Library Configuration 120
OSLIBRARY_GLOBALS: Specify Memory Type for Global Salvo Objects in Precompiled Library 121
OSLIBRARY_TYPE: Specify Precompiled Library Type 122
OSLIBRARY_VARIANT: Specify Precompiled Library Variant 123

OSMESSAGE_QUEUES: Set Maximum Number of Message Queues 124
OSTARGET: Identify Target Processor 125
OSTASKS: Set Maximum Number of Tasks 126
OSUSE_LIBRARY: Use Precompiled Library 127
Configuration Options for Source Code Distributions 130
OSBIG_MESSAGE_POINTERS: Allow Message Pointers to RAM and ROM 131
OSBIG_SEMAPHORES: Use 16-bit Semaphores 132
OSBYTES_OF_COUNTS: Set Size of Counters 133
OSBYTES_OF_EVENT_FLAGS: Set Size of Event Flags 134
OSBYTES_OF_DELAYS: Set Length of Delays 135
OSBYTES_OF_TICKS: Set Maximum System Tick Count 136
OSCALL_OSCREATEEVENT: Manage Interrupts when Creating Events 137
OSCALL_OSGETPRIOTASK: Manage Interrupts when Returning a Task's Priority 140
OSCALL_OSGETSTATETASK: Manage Interrupts when Returning a Task's State 140
OSCALL_OSMSGQCOUNT: Manage Interrupts when Returning Number of Messages in Message Queue 140
OSCALL_OSMSGQEMPTY: Manage Interrupts when Checking if Message Queue is Empty 140
OSCALL_OSRETURNEVENT: Manage Interrupts when Reading and/or Trying Events 141
OSCALL_OSSIGNALEVENT: Manage Interrupts when Signaling Events and Manipulating Event Flags 141
OSCALL_OSSTARTTASK: Manage Interrupts when Starting Tasks 141
OSCLEAR_GLOBALS: Explicitly Clear all Global Parameters 142
OSCLEAR_UNUSED_POINTERS: Reset Unused Tcb and Ecb Pointers 143
OSCLEAR_WATCHDOG_TIMER(): Define Instruction(s) to Clear the Watchdog Timer 144
OSCOMBINE_EVENT_SERVICES: Combine Common Event Service Code 145
OSCTXSW_METHOD: Identify Context-Switching Methodology in Use 147
OSDISABLE_ERROR_CHECKING: Disable Runtime Error Checking 148
OSDISABLE_FAST_SCHEDULING: Configure Round-Robin Scheduling 149
OSDISABLE_TASK_PRIORITIES: Force All Tasks to Same Priority 150
OSENABLE_BINARY_SEMAPHORES: Enable Support for Binary Semaphores 151
OSENABLE_BOUNDS_CHECKING: Enable Runtime Pointer Bounds Checking 152
OSENABLE_EVENT_FLAGS: Enable Support for Event Flags 153
OSENABLE_EVENT_READING: Enable Support for Event Reading 154
OSENABLE_EVENT_TRYING: Enable Support for Event Trying 155
OSENABLE_FAST_SIGNALING: Enable Fast Event Signaling 156
OSENABLE_IDLE_COUNTER: Track Scheduler Idling 157
OSENABLE_IDLING_HOOK: Call a User Function when Idling 158
OSENABLE_INTERRUPT_HOOKS: Call User Functions when Controlling Interrupts 159
OSENABLE_MESSAGES: Enable Support for Messages 161
OSENABLE_MESSAGE_QUEUES: Enable Support for Message Queues 162
OSENABLE_SCHEDULER_HOOK: Call User Function Inside Scheduler 163
OSENABLE_SEMAPHORES: Enable Support for Semaphores 164
OSENABLE_STACK_CHECKING: Monitor Call ... Return Stack Depth 165
OSENABLE_TCBEXT0|1|2|3|4|5: Enable Tcb Extensions 166
OSENABLE_TIMEOUTS: Enable Support for Timeouts 169
OSGATHER_STATISTICS: Collect Run-time Statistics 170
OSINTERRUPT_LEVEL: Specify Interrupt Level for Interrupt-callable Services 171

OSLOC_ALL: Storage Type for All Salvo Objects 172
OSLOC_COUNT: Storage Type for Counters 174
OSLOC_CTCB: Storage Type for Current Task Control Block Pointer 175
OSLOC_DEPTH: Storage Type for Stack Depth Counters 175
OSLOC_ECB: Storage Type for Event Control Blocks and Queue Pointers 175
OSLOC_EFCB: Storage Type for Event Flag Control Blocks 175
OSLOC_ERR: Storage Type for Error Counters 176
OSLOC_GLSTAT: Storage Type for Global Status Bits 176
OSLOC_LOGMSG: Storage Type for Log Message String 176
OSLOC_MQCB: Storage Type for Message Queue Control Blocks 176
OSLOC_MSGQ: Storage Type for Message Queues 177
OSLOC_PS: Storage Type for Timer Prescalar 177
OSLOC_TCB: Storage Type for Task Control Blocks 177
OSLOC_SIGQ: Storage Type for Signaled Events Queue Pointers 178
OSLOC_TICK: Storage Type for System Tick Counter 178
OSLOGGING: Log Runtime Errors and Warnings 179
OSLOG_MESSAGES: Configure Runtime Logging Messages 180
OSMPLAB_C18_LOC_ALL_NEAR: Locate all Salvo Objects in Access Bank (MPLAB-C18 Only) 182
OSOPTIMIZE_FOR_SPEED: Optimize for Code Size or Speed 183
OSPIC18_INTERRUPT_MASK: Configure PIC18 Interrupt Mode 184
OSPRESERVE_INTERRUPT_MASK: Control Interrupt-enabling Behavior 186
OSRPT_HIDE_INVALID_POINTERS: OSRpt() Won't Display Invalid Pointers 187
OSRPT_SHOW_ONLY_ACTIVE: OSRpt() Displays Only Active Task and Event Data 188
OSRPT_SHOW_TOTAL_DELAY: OSRpt() Shows the Total Delay in the Delay Queue 189
OSRTNADDR_OFFSET: Offset (in bytes) for Context-Switching Saved Return Address 190
OSSCHED_RETURN_LABEL(): Define Label within OSSched() 191
OSSET_LIMITS: Limit Number of Runtime Salvo Objects 192
OSSPEEDUP_QUEUEING: Speed Up Queue Operations 193
OSTIMER_PRESCALAR: Configure Prescalar for OSTimer() 194
OSTYPE_TCBEXT0|1|2|3|4|5: Set Tcb Extension Type 195
OSUSE_CHAR_SIZED_BITFIELDS: Pack Bitfields into Chars 196
OSUSE_EVENT_TYPES: Check for Event Types at Runtime 197
OSUSE_INLINE_OSSCHED: Reduce Task Call…Return Stack Depth 198
OSUSE_INLINE_OSTIMER: Eliminate OSTimer() Call…Return Stack Usage 200
OSUSE_INSELIG_MACRO: Reduce Salvo's Call Depth 201
OSUSE_MEMSET: Use memset() (if available) 202
Other Symbols 203
MAKE_WITH_FREE_LIB, MAKE_WITH_STD_LIB: Use salvocfg.h for Multiple Projects 203
SYSA|B|…|Z: Identify Salvo Test System 205
USE_INTERRUPTS: Enable Interrupt Code 207
Obsolete Configuration Parameters 209
As of 3.1.0 209

Chapter 6 • Frequently Asked Questions (FAQ) 211

General 211

What is Salvo? 211
Is there a shareware / freeware / open source version of Salvo? 211
Just how small is Salvo? 212
Why should I use Salvo? 212
What can I do with Salvo? 213
What kind of RTOS is Salvo? 213
What are Salvo's minimum requirements? 213
What kind of processors can Salvo applications run on? 214
How many tasks and events does Salvo support? 214
How many priority levels does Salvo support? 214
What kind of events does Salvo support? 214
Is Salvo Y2K compliant? 215
Where did Salvo come from? 215
Getting Started 215
Where can I find examples of projects that use Salvo? 215
Is there a tutorial? 215
Apart from the Salvo User Manual, what other sources of documentation are available? 215
I'm on a tight budget. Can I use Salvo? 215
I only have an assembler. Can I use Salvo? 216
Performance 216
How can using Salvo improve the performance of my application? 216
How do delays work under Salvo? 217
What's so great about having task priorities? 217
When does the Salvo code in my application actually run? 217
How can I perform fast, timing-critical operations under Salvo? 218
Memory 218
How much will Salvo add to my application's ROM and RAM usage? 218
How much RAM will an application built with the libraries use? 219
Do I need to worry about running out of memory? 219
If I define a task or event but never use it, is it costing me RAM? 220
How much call ... return stack depth does Salvo use? 220
Why must I use pointers when working with tasks? Why can't I use explicit task IDs? 220
How can I avoid re-initializing Salvo's variables when I wake up from sleep on a PIC12C509 PICmicro MCU? 222
Libraries 222
What kinds of libraries does Salvo include? 222
What's in each Salvo library? 222
Why are there so many libraries? 223
Should I use the libraries or the source code when building my application? 223
What's the difference between the freeware and standard Salvo libraries? 223
My library-based application is using more RAM than I can account for. Why? 223
I'm using a library. Why does my application use more RAM than one compiled directly from source files? 223
I'm using a freeware library and I get the message "#error: OSXYZ exceeds library limit – aborting." Why? 224
I'm using a standard library and I can't increase the number of tasks beyond the library's default. Why? 224
Why can't I alter the functionality of a library by adding configuration options to my salvocfg.h? 224

The libraries are very large – much larger than the ROM size of my target processor. Won't that affect my application? 225
Why is there a precompiled mem.c object in each library? 225
I'm using a library. Can I change the bank where Salvo variables are located? 225
Configuration 225
I'm overwhelmed by all the configuration options. Where should I start? 225
Do I have to use all of Salvo's functionality? 226
What file(s) do I include in my main.c? 226
What is the purpose of OSENABLE_SEMAPHORES and similar configuration options? 226
Can I collect run-time statistics with Salvo? 226
How can I clear my processor's watchdog timer with Salvo? 227
I enabled timeouts and my RAM and ROM grew substantially– why? 227
Timer and Timing 227
Do I have to install the timer? 227
How do I install the timer? 228
I added the timer to my ISR and now my ISR is huge and slow. What should I do? 228
How do I pick a tick rate for Salvo? 228
How do I use the timer prescalar? 228
I enabled the prescalar and set it to 1 but it didn't make any difference. Why? 229
What is the accuracy of the system timer? 229
What is Salvo's interrupt latency? 229
What if I need to specify delays larger than 8 bits of ticks? 229
How can I achieve very long delays via Salvo? Can I do that and still keep task memory to a minimum? 230
Can I specify a timeout when waiting for an event? 231
Does Salvo provide functions to obtain elapsed time? 231
How do I choose the right value for OSBYTES_OF_TICKS? 231
My processor has no interrupts. Can I still use Salvo's timer services? 232
Context Switching 232
How do I know when I'm context switching in Salvo? 232
Why can't I context switch from something other than the task level? 232
Why does Salvo use macros to do context switching? 233
Can I context switch in more than one place per task? 233
When must I use context-switching labels? 233
Tasks & Events 234
What are taskIDs? 234
Does it matter which taskID I assign to a particular task? 234
Is there an idle task in Salvo? 234
How can I monitor the tasks in my application? 235
What exactly happens in the scheduler? 235
What about reentrant code and Salvo? 235
What are "implicit" and "explicit" OS task functions? 235
How do I setup an infinite loop in a task? 236
Why must tasks use static variables? 236
Can tasks share the same priority? 237
Can I have multiple instances of the same task? 237
Does the order in which I start tasks matter? 237
How can I reduce code size when starting tasks? 238
What is the difference between a delayed task and a waiting task? 238
Can I create a task to immediately wait an event? 239

I started a task but it never ran. Why? 239
What happens if I forget to loop in my task? 239
Why did my low-priority run-time tasks start running before my high-priority startup task completed? 240
When I signaled a waiting task, it took much longer than the context switching time to run. Why? 240
Can I destroy a task and (re-) create a new one in its place? 240
Can more than one task wait on an event? 241
Does Salvo preserve the order in which events occur? 241
Can a task wait on more than one event at a time? 241
How can I implement event flags? 242
What happens when a task times out waiting for an event? 243
Why is my high-priority task stuck waiting, while other low-priority tasks are running? 243
When an event occurs and there are tasks waiting for it, which task(s) become eligible? 243
How can I tell if a task timed out waiting for an event? 244
Can I create an event from inside a task? 244
What kind of information can I pass to a task via a message? 245
My application uses messages and binary semaphores. Is there any way to make the Salvo code smaller? 245
Why did RAM requirements increase substantially when I enabled message queues? 245
Can I signal an event from outside a task? 246
When I signal a message that has more than one task waiting for it, why does only one task become eligible? 246
I'm using a message event to pass a character variable to a waiting task, but I don't get the right data when I dereference the pointer. What's going on? 246
What happens when there are no tasks in the eligible queue? 247
In what order do messages leave a message queue? 247
What happens if an event is signaled before any task starts to wait it? Will the event get lost or it will be processed after task starts to wait it? 248
What happens if an event is signaled several times before waiting task gets a chance to run and process that event? Will the last one signal be processed and previous lost? Or the first will be processed and the following signals lost? 248
What is more important to create first, an event or the task that waits it? Does the order of creation matter? 248
What if I don't need one event anymore and want to use its slot for another event? Can I destroy event? 248
Can I use messages or message queues to pass raw data between tasks? 249
How can I test if there's room for additional messages in a message queue without signaling the message queue? 249
Interrupts 249
Why does Salvo disable all interrupts during a critical section of code? 249
Can I modify Salvo to disable only certain interrupts during critical sections of code? 250
How big are the Salvo functions I might call from within an interrupt? 250
Why did my interrupt service routine grow and become slower when I added a call to OSTimer()? 250
My application can't afford the overhead of signaling from an ISR. How can I get around this problem? 251
Building Projects 252
What warning level should I use when building Salvo projects? 252

What optimization level should I use when building Salvo projects? 252
Miscellaneous 252
Can Salvo run on a 12-bit PICmicro with only a 2-level call…return stack? 252

Chapter 7 • Reference 253

User Services 253
Macro Declarations 254
Function Prototypes 255
OS_Delay(): Delay the Current Task and Context-switch 258
OS_DelayTS(): Delay the Current Task Relative to its Timestamp and Context-switch 260
OS_Destroy(): Destroy the Current Task and Context-switch 262
OS_Replace(): Replace the Current Task and Context-switch 264
OS_SetPrio(): Change the Current Task's Priority and Context-switch 266
OS_Stop(): Stop the Current Task and Context-switch 268
OS_WaitBinSem(): Context-switch and Wait the Current Task on a Binary Semaphore 270
OS_WaitEFlag(): Context-switch and Wait the Current Task on an Event Flag 272
OS_WaitMsg(): Context-switch and Wait the Current Task on a Message 276
OS_WaitMsgQ(): Context-switch and Wait the Current Task on a Message Queue 278
OS_WaitSem(): Context-switch and Wait the Current Task on a Semaphore 280
OS_Yield(): Context-switch 282
OSClrEFlag(): Clear Event Flag Bit(s) 284
OSCreateBinSem(): Create a Binary Semaphore 286
OSCreateEFlag(): Create an Event Flag 288
OSCreateMsg(): Create a Message 290
OSCreateMsgQ(): Create a Message Queue 292
OSCreateSem(): Create a Semaphore 294
OSCreateTask(): Create and Start a Task 296
OSDestroyTask(): Destroy a Task 298
OSGetPrio(): Return the Current Task's Priority 300
OSGetPrioTask(): Return the Specified Task's Priority 302
OSGetState(): Return the Current Task's State 304
OSGetStateTask(): Return the Specified Task's State 306
OSGetTicks(): Return the System Timer 308
OSGetTS(): Return the Current Task's Timestamp 310
OSInit(): Prepare for Multitasking 312
OSMsgQCount(): Return Number of Messages in Message Queue 314
OSMsgQEmpty(): Check for Available Space in Message Queue 316
OSReadBinSem(): Obtain a Binary Semaphore Unconditionally 318
OSReadEFlag(): Obtain an Event Flag Unconditionally 320
OSReadMsg():Obtain a Message's Message Pointer Unconditionally 322
OSReadMsgQ(): Obtain a Message Queue's Message Pointer Unconditionally 324
OSReadSem(): Obtain a Semaphore Unconditionally 326
OSRpt(): Display the Status of all Tasks, Events, Queues and Counters 328
OSSched(): Run the Highest-Priority Eligible Task 330
OSSetEFlag(): Set Event Flag Bit(s) 332
OSSetPrio(): Change the Current Task's Priority 334
OSSetPrioTask(): Change a Task's Priority 336
OSSetTicks(): Initialize the System Timer 338

OSSetTS(): Initialize the Current Task's Timestamp....340
OSSignalBinSem(): Signal a Binary Semaphore....342
OSSignalMsg(): Send a Message....344
OSSignalMsgQ(): Send a Message via a Message Queue....346
OSSignalSem(): Signal a Semaphore....348
OSStartTask(): Make a Task Eligible To Run....350
OSStopTask(): Stop a Task....352
OSSyncTS(): Synchronize the Current Task's Timestamp....354
OSTimer(): Run the Timer....356
OSTryBinSem(): Obtain a Binary Semaphore if Available....358
OSTryMsg(): Obtain a Message if Available....360
OSTryMsgQ(): Obtain a Message from a Message Queue if Available....362
OSTrySem(): Obtain a Semaphore if Available....364
Additional User Services....366
OSAnyEligibleTasks (): Check for Eligible Tasks....366
OScTcbExt0|1|2|3|4|5, OStcbExt0|1|2|3|4|5(): Return a Tcb Extension....368
OSDi(), OSEi(): Control Interrupts....370
OSProtect(), OSUnprotect(): Protect Services Against Corruption by ISR....372
OSTimedOut(): Check for Timeout....374
OSVersion(), OSVERSION: Return Version as Integer....376
User Macros....378
_OSLabel(): Define Label for Context Switch....378
OSECBP(), OSEFCBP(),OSMQCBP(), OSTCBP(): Return a Control Block Pointer....380
User-Defined Services....382
OSDisableIntsHook(), OSEnableIntsHook(): Interrupt-control Hooks....382
OSIdlingHook(): Idle Function Hook....384
Return Codes....386
Salvo Defined Types....386
Salvo Variables....390
Salvo Source Code....391
Locations of Salvo Functions....393
Abbreviations Used by Salvo....395

Chapter 8 • Libraries....399

Library Types....399
Libraries for Different Environments....399
Native Compilers....399
Non-native Compilers....400
Using the Libraries....400
Overriding Default RAM Settings....401
Library Functionality....402
Types....403
Memory Models....403
Options....404
Global Variables....404
Configurations....404
Variants....405
Library Reference....408
Microchip PICmicro MCUs: HI-TECH PICC Compiler....408
Compiler and Target Processor Configuration Options....408

Nomenclature 408
Example salvocfg.h 409
Build Settings 409
Processors 410
Notes 412
Microchip PICmicro MCUs: HI-TECH PICC-18 C Compiler 413
Compiler and Target Processor Configuration Options 413
Nomenclature 413
Example salvocfg.h 413
Build Settings 414
Processors 415
Notes 415
Microchip PIC18 PICmicro MCUs: IAR PIC18 C Compiler 417
Compiler and Target Processor Configuration Options 417
Nomenclature 417
Example salvocfg.h 418
Build Settings 418
Processors 419
Notes 420
Microchip PIC18 PICmicro MCUs: MPLAB-C18 C Compiler 421
Compiler and Target Processor Configuration Options 421
Nomenclature 421
Example salvocfg.h 421
Build Settings 422
Processors 423
Notes 424
8051 Family: HI-TECH 8051C C Compiler 425
Compiler and Target Processor Configuration Options 425
Nomenclature 425
Example salvocfg.h 426
Build Settings 426
Notes 427
8051 Family: Keil Cx51 C Compiler 428
Compiler and Target Processor Configuration Options 428
Nomenclature 428
Example salvocfg.h 429
Build Settings 429
Notes 430
Motorola M68HC11: ImageCraft ICC11 C Compiler 431
Compiler and Target Processor Configuration Options 431
Nomenclature 431
Example salvocfg.h 431
Build Settings 432
Processors 433
Notes 433
TI's MSP430: Archelon / Quadravox AQ430 C Compiler 434
Compiler and Target Processor Configuration Options 434
Nomenclature 434
Example salvocfg.h 434
Build Settings 435
Processors 436

Notes....436
TI's MSP430: IAR MSP430 C Compiler....437
Compiler and Target Processor Configuration Options....437
Nomenclature....437
Example salvocfg.h....437
Build Settings....438
Processors....438
Notes....439
TI's MSP430: ImageCraft ICC430 C Compiler....440
Compiler and Target Processor Configuration Options....440
Nomenclature....440
Example salvocfg.h....440
Build Settings....441
Processors....442
Notes....442
ARC / VAutomation V8-µRISC: HI-TECH V8C C Compiler....443
Compiler and Target Processor Configuration Options....443
Nomenclature....443
Example salvocfg.h....443
Build Settings....444
Notes....444
x86 PCs: GNU C Compiler (gcc)....446
Compiler and Target Processor Configuration Options....446
Nomenclature....446
Example salvocfg.h....446
Build Configuration....447
Notes....447
GNU Public License (GPL) Issues....447
x86 PCs: Metrowerks CodeWarrior C Compiler....449
Compiler and Target Processor Configuration Options....449
Nomenclature....449
Example salvocfg.h....449
Build Configuration....450
Notes....450
Rebuilding the Libraries....451
Customizing the Libraries....451
Linux/Unix Environment....452
Win32 Environment....452
Makefile Descriptions....453
salvo\src\Makefile....453
salvo\src\Makefile2....453
salvo\src\targets.mk....453
salvo\src\makeXyz.bat....453

Chapter 9 • Performance....455

Introduction....455
Measuring Performance....455
Performance Examples....455
Test Systems....456
Test Configurations....457

Test Programs 457
Compile-time Performance 459
Code Size (ROM) 459
Variables (RAM) 461
Run-time Performance 462
Speeds of User Services 463
OS_Delay() 464
OS_Destroy() 464
OS_Prio() 464
OS_Stop() 464
OS_WaitBinSem() 464
OS_WaitMsg() 465
OS_WaitMsgQ() 465
OS_WaitSem() 465
OS_Yield() 465
OSCreateBinSem() 466
OSCreateMsg() 466
OSCreateMsgQ() 466
OSCreateSem() 466
OSCreateTask() 466
OSInit() 466
OSSched() 467
OSSignalBinSem() 468
OSSignalMsg() 468
OSSignalMsgQ() 468
OSSignalSem() 468
OSStartTask() 469
OSTimer() 469
Maximum Variable Execution Times 469
t_InsPrioQ 470
t_DelPrioQ 470
t_InsDelayQ 470
t_DelDelayQ 471
Impact of Queueing Operations 472
Simple Queues 475
t_InsPrioQ 475
Configurations I & III 475
Configurations II & IV 475
Configuration V 476
t_DelPrioQ 476
Configurations I & III 476
Configurations II & IV 476
Configuration V 476
t_InsDelayQ 476
Configurations II & IV 477
Configuration V 477
t_DelDelayQ 478
Configurations II & IV 478
Configuration V 479
Other Variable-speed Operations 479
t_InitTcb 479

Configuration I 480
Configuration II 480
Configuration III 480
Configuration IV 480
Configuration V 480
t_InitEcb 480
Configuration I 481
Configuration II 481
Configuration III 481
Configuration IV 481
Configuration V 481

Chapter 10 • Porting 483

Introduction 483
Compiler-Specific Issues in the Source Code 483
Compiling the Salvo Source Code 484
Controlling Interrupts 485
HI-TECH PICC Compiler 486
Mix Power C Compiler 486
Metrowerks CodeWarrior Compiler 486
Context Switching 486
HI-TECH PICC Compiler 487
Mix Power C Compiler 487
Metrowerks CodeWarrior Compiler 488
Context-switching Label Declarator 488
HI-TECH PICC Compiler 488
Mix Power C Compiler 489
Metrowerks CodeWarrior Compiler 490
Managing RAM 490
Pointers 490
Linking the Salvo Source Code 491
Executing the Salvo Source Code 491
Optimizing and Saving Registers 492
Verifying the Required Stack Depth 492
Characterizing RAM and ROM Requirements 492
Characterizing Run-time Performance 493

Chapter 11 • Tips, Tricks and Troubleshooting 495

Introduction 495
Compile-Time Troubleshooting 496
I'm just starting, and I'm getting lots of errors. 496
My compiler can't find salvo.h. 496
My compiler can't find salvocfg.h. 496
My compiler can't find certain target-specific header files. 496
My compiler can't locate a particular Salvo service. 496
My compiler has issued an "undefined symbol" error for a context-switching label that I've defined properly. 497
My compiler is saying something about OSIdlingHook. 497
My compiler has no command-line tools. Can I still build a library? 497

Run-Time Troubleshooting 498
Nothing's happening. 498
It only works if I single-step through my program. 499
It still doesn't work. How should I begin debugging? 499
My program's behavior still doesn't make any sense. 500
Compiler Issues 500
Where can I get a free C compiler? 500
Where can I get a free make utility? 501
Where can I get a Linux/Unix-like shell for my Windows PC? 501
My compiler behaves strangely when I'm compiling from the DOS command line, e.g. "This program has performed an illegal operation and will be terminated." 501
My compiler is issuing redeclaration errors when I compile my program with Salvo's source files. 502
HI-TECH PICC Compiler 502
Running HPDPIC under Windows 2000 Pro 502
Setting PICC Error/Warning Format under Windows 2000 Pro 503
Linker reports fixup errors 503
Placing variables in RAM 504
Link errors when working with libraries 504
Avoiding absolute file pathnames 504
Compiled code doesn't work 505
PIC17CXXX pointer passing bugs 505
While() statements and context switches 505
Library generation in HPDPIC 505
Problems banking Salvo variables on 12-bit devices 506
Working with Salvo messages 506
Adding OSTimer() to an Interrupt Service Routine 506
Using the interrupt_level pragma 508
HI-TECH V8C Compiler 508
Simulators 508
HI-TECH 8051C Compiler 509
Problems with static initialization and small and medium memory models. 509
IAR PICC Compiler 509
Target-specific header files 509
Interrupts 509
Mix Power C Compiler 510
Required compile options 510
Application crashes after adding long C source lines to a Salvo task 511
Application crashes after adding complex expressions to a Salvo task 511
Application crashes when compiling with /t option 512
Compiler crashes when using a make system 512
Metrowerks CodeWarrior Compiler 512
Compiler has a fatal internal error when compiling your source code 512
Microchip MPLAB 513
The Stack window shows nested interrupts 513
Controlling the Size of your Application 513
Working with Message Pointers 514

Appendix A • Recommended Reading 517

Salvo Publications 517

Application Notes 517
Assembly Guides 517
Learning C 518
K&R 518
C, A Reference Manual 518
Power C 518
Real-time Kernels 518
µC/OS & MicroC/OS-II 518
CTask 519
Embedded Programming 519
RTOS Issues 519
Priority Inversions 520
Microcontrollers 520
PIC16 520

Appendix B • Other Resources 521

Web Links to Other Resources 521

Appendix C • File and Program Descriptions 523

Overview 523
Test Systems 523
Projects 525
Nomenclature 525
Source Files 526
SYS Predefined Symbols 526
File Types 527
Included Projects and Programs 530
Demonstration Programs 530
demo\d1\sysa|e|f|t 530
demo\d2\sysa|f|h 530
demo\d3\sysa|j † 531
demo\d4\sysa|e|f|h † 531
Example Programs 532
ex\ex1\sysa|e|f|h|p|q|r|s|t 532
ex\ex2\sysa 532
Templates 532
tplt\te1 532
Test Programs 533
test\t1\sysa|b|c|d 533
test\t2\sysa|b|c|d 533
test\t3\sysa|b|c|d 533
test\t4\sysa|b|c 533
test\t5\sysa|b|c 534
test\t6\sysa|b|c|d 534
test\t7\sysa|b|c|d 534
test\t8\sysa|b|c|d 534
test\t9\sysa|b|c|d 535
test\t10\sysa|b|c|d 535
test\t11\sysa 535

test\t12\sysa 535
test\t13\sysa 535
test\t14\sysa 535
test\t15\sysa 536
test\t16\sysa 536
test\t17\sysa 536
test\t18\sysa 536
test\t19\sysa 536
test\t20\sysa 536
test\t21\sysa 536
test\t22\sysa 537
test\t23\sysa 537
test\t24\sysa 537
test\t25\sysa 537
test\t26\sysa 537
test\t27\sysa 537
test\t28\sysa 537
test\t29\sysa 537
test\t30\sysa 538
test\t31\sysa 538
test\t32\sysa 538
test\t33\sysa 538
test\t34\syse|f 538
test\t35\syso 538
test\t36\sysa 538
test\t37\sysf 538
test\t38 539
test\t39 539
test\t40-t47\sysa|e|f|l|p|q|r|s|t 539
Tutorial Programs 540
tut\tu1\sysa|e|f|h|i|l|q|r|s|t † 540
tut\tu2\sysa|e|f|h|i|l|q|r|s|t † 541
tut\tu3\sysa|e|f|h|i|l|q|r|s|t † 541
tut\tu4\sysa|e|f|h|i|l|q|r|s|t † 541
tut\tu5\sysa|e|f|h|i|l|q|r|s|t † 541
tut\tu6\sysa|e|f|h|i|l|q|r|s|t † 541
Library Files 542
lib*.* 542
Third-Party Files 542
free\links*.* 542

Index 543

Figures

Figure 1: Foreground / Background Processing 14
Figure 2: Interrupts Can Occur While Tasks Are Running 18
Figure 3: Preemptive Scheduling 19
Figure 4: Cooperative Scheduling 20
Figure 5: Task States 23
Figure 6: Binary and Counting Semaphores 29
Figure 7: Signaling a Binary Semaphore 30
Figure 8: Waiting a Binary Semaphore When the Event Has Already Occurred 30
Figure 9: Signaling a Binary Semaphore When a Task is Waiting for the Corresponding Event 31
Figure 10: Synchronizing Two Tasks with Event Flags 32
Figure 11: Using a Counting Semaphore to Implement a Ring Buffer 34
Figure 12: Signaling a Message with a Pointer to the Message's Contents 36
Figure 13: Welcome Screen 52
Figure 14: Important Notes Screen 52
Figure 15: Salvo License Agreement Screen 53
Figure 16: Registration Screen 54
Figure 17: Choose Destination Location Screen 54
Figure 18: Setup Type Screen 55
Figure 19: Ready To Install Screen 56
Figure 20: Finished Screen 56
Figure 21: Typical Salvo Destination Directory Contents 58
Figure 22: Start Menu Programs Folder 58
Figure 23: Launching the Uninstaller 59
Figure 24: Confirm File Deletion Screen 59
Figure 25: Uninstall Complete Screen 59
Figure 26: Creating the Tutorial Project 92
Figure 27: Selecting the Processor Type 92
Figure 28: Selecting the Float Type 93
Figure 29: Selecting the Output File Format 93
Figure 30: Selecting the Optimizations 94
Figure 31: Selecting the Map and Symbol File Options 94
Figure 32: Adding the Source File main.c 95
Figure 33: Saving the Project 95
Figure 34: Setting the Include Paths 97
Figure 35: Link Errors due to Missing Source Files 98
Figure 36: The Library File List 99
Figure 37: Freeware Library at Head of Library File List 100
Figure 38: Adding the Obvious Source Files 101
Figure 39: Link Errors due to Missing Source Files 102
Figure 40: Complete List of Required Source Files 102
Figure 41: A Successful Compilation 103
Figure 42: Display of Memory Usage 104
Figure 43: Creating a New Project in MPLAB 106
Figure 44: Accepting Project Defaults in MPLAB 106

Figure 45: Downloading the Program 107
Figure 46: Symbolic Debugging via MPLAB 107
Figure 47: How to call OSCreateBinSem() when OSCALL_OSCREATEEVENT is set to OSFROM_BACKGROUND 138
Figure 48: How to call OSCreateBinSem() when OSCALL_OSCREATEBINSEM is set to OSFROM_FOREGROUND 138
Figure 49: How to call OSCreateBinSem() when OSCALL_CREATEBINSEM is set to OSFROM_ANYWHERE 139
Figure 50: Tcb Extension Example Program Output 168
Figure 51: OSRpt() Output to Terminal Screen 329
Figure 52: Salvo Library Nomenclature – HI-TECH PICC C Compiler 408
Figure 53: Salvo Library Nomenclature – HI-TECH PICC-18 C Compiler 413
Figure 54: Salvo Library Nomenclature – IAR PIC18C C Compiler 417
Figure 55: Salvo Library Nomenclature – Microchip MPLAB-C18 C Compiler 421
Figure 56: Salvo Library Nomenclature – HI-TECH 8051C C Compiler 425
Figure 57: Salvo Library Nomenclature – Keil Cx51 C Compiler 428
Figure 58: Salvo Library Nomenclature – ImageCraft ICC11 C Compiler 431
Figure 59: Salvo Library Nomenclature – Archelon AQ430 C Compiler 434
Figure 60: Salvo Library Nomenclature – IAR MSP430 C Compiler 437
Figure 61: Salvo Library Nomenclature – ImageCraft ICC430 C Compiler 440
Figure 62: Salvo Library Nomenclature – HI-TECH V8C C Compiler 443
Figure 63: Salvo Library Nomenclature – GNU C Compiler 446
Figure 64: Salvo Library Nomenclature – Metrowerks CodeWarrior C Compiler 449

Listings

Listing 1: A Simple Salvo Program 4
Listing 2: C Compiler Feature Requirements 7
Listing 3: Reentrancy Errors with printf() 15
Listing 4: Task Structure for Preemptive Multitasking 21
Listing 5: Task Structure for Cooperative Multitasking 22
Listing 6: Delay Loop 24
Listing 7: Delaying via the RTOS 26
Listing 8: Examples of Events 27
Listing 9: Task Synchronization with Binary Semaphores 32
Listing 10: Using a Binary Semaphore to Control Access to a Resource 33
Listing 11: Using a Counting Semaphore to Control Access to a Resource 35
Listing 12: Signaling a Message with a Pointer 36
Listing 13: Receiving a Message and Operating on its Contents 37
Listing 14: Vending Machine Superloop 40
Listing 15: Task Version of ReleaseItem() 43
Listing 16: Task Version of CallPolice() 44
Listing 17: Prioritizing a Task 44
Listing 18: Creating a Message Event 45
Listing 19: Calling the System Timer 45
Listing 20: Starting all Tasks 45
Listing 21: Multitasking Begins 46
Listing 22: RTOS-based Vending Machine 48
Listing 23: A Minimal Salvo Application 62
Listing 24: A Multitasking Salvo Application with two Tasks 63
Listing 25: Multitasking with two Non-trivial Tasks 67
Listing 26: Multitasking with an Event 69
Listing 27: Multitasking with a Delay 74
Listing 28: Calling OSTimer() at the System Tick Rate 74
Listing 29: Signaling from Multiple Tasks 77
Listing 30: salvocfg.h for Tutorial Program 97
Listing 31: Tcb Extension Example 167
Listing 32: salvocfg.h for Multiple Projects 204
Listing 33: Use of SYSA … SYSZ in main.c 205
Listing 34: Use of SYSA … SYSZ in salvocfg.h 206
Listing 35: Use of USE_INTERRUPTS in isr.c 208
Listing 36: Obsolete Configuration Parameters 209
Listing 37: Source Code Files 392
Listing 38: Target-specific Source and Header Files 393
Listing 39: Location of Functions in Source Code 395
Listing 40: List of Abbreviations 396
Listing 41: Example salvocfg.h for Use with Standard Library 401
Listing 42: Example salvocfg.h for Use with Standard Library and Reduced Number of Tasks 402
Listing 43: Additional Lines in salvocfg.h for Reducing Memory Usage with Salvo Libraries . 402
Listing 44: Partial Listing of Services than can be called from Interrupts 406
Listing 45: Example salvocfg.h for Library Builds – HI-TECH PICC C Compiler 409

Listing 46: Setting the HI-TECH PICC interrupt_level pragma for an ISR when Using a-variant Libraries 412
Listing 47: Example salvocfg.h for Library Builds – HI-TECH PICC-18 C Compiler 414
Listing 48: Setting the HI-TECH PICC-18 interrupt_level pragma for an ISR when Using a-variant Libraries 416
Listing 49 Example salvocfg.h for Library Builds – IAR PIC18C C Compiler 418
Listing 50 Example salvocfg.h for Library Builds – Microchip MPLAB-C18 C Compiler 422
Listing 51: Example salvocfg.h for Library Builds – HI-TECH 8051C C Compiler 426
Listing 52: Example salvocfg.h for Library Builds – Keil Cx51 C Compiler 429
Listing 53 Example salvocfg.h for Library Builds – ImageCraft ICC11 C Compiler 432
Listing 54 Example salvocfg.h for Library Builds – Archelon AQ430 C Compiler 435
Listing 55 Example salvocfg.h for Library Builds – IAR MSP430 C Compiler 437
Listing 56 Example salvocfg.h for Library Builds – ImageCraft ICC430 C Compiler 441
Listing 57: Example salvocfg.h for Library Builds – HI-TECH V8C C Compiler 444
Listing 58: Example salvocfg.h Library Builds – GNU C Compiler 446
Listing 59: Linking a Salvo Application to a Salvo Library for the GNU C Compiler 447
Listing 60: Example salvocfg.h for use with Salvo Libraries – Metrowerks CodeWarrior C Compiler 450
Listing 61: Rebuilding Salvo Libraries in a Linux/Unix Environment 452
Listing 62: Obtaining a List of Library Targets in the Makefile 452
Listing 63: Building the Salvo PICC Libraries for mid-range PICmicros in the Win32 Environment 454
Listing 64: Sample salvocfg.h for Porting Testing 484
Listing 65: Sample Salvo Application for Porting Testing 485

Tables

Table 1: Supported Targets and Compilers xxix
Table 2: Configuration Options by Category 111
Table 3: Configuration Options by Desired Feature 113
Table 4: Predefined Symbols 115
Table 5: Allowable Storage Types / Type Qualifiers for Salvo Objects 173
Table 6: Return Codes 386
Table 7: Normal Types 388
Table 8: Normal Pointer Types 388
Table 9: Qualified Types 389
Table 10: Qualified Pointer Types 389
Table 11: Salvo Variables 391
Table 12: Type Codes for Salvo Libraries 403
Table 13: Configuration Codes for Salvo Libraries 404
Table 14: Features Common to all Salvo Library Configurations 405
Table 15: Variant Codes for Salvo Libraries 407
Table 16: Configuration Options Used to Build Salvo Libraries – HI-TECH PICC C Compiler 410
Table 17: Processors for Salvo Libraries – HI-TECH PICC C Compiler 411
Table 18: Configuration Options Used to Build Salvo Libraries – HI-TECH PICC-18 C Compiler 414
Table 19: Processors for Salvo Libraries – HI-TECH PICC-18 C Compiler 415
Table 20: Configuration Options Used to Build Salvo Libraries – IAR PIC18C C Compiler 418
Table 21: Memory Model Codes for Salvo Libraries – IAR PIC18 C Compiler 419
Table 22: Global Salvo Object Codes for Salvo Libraries – IAR PIC18 C Compiler 419
Table 23: Processor Types for Salvo Libraries – IAR PIC18 C Compiler 420
Table 24: Configuration Options Used to Build Salvo Libraries – Microchip MPLAB-C18 C Compiler 422
Table 25: Memory Model Codes for Salvo Libraries – Microchip MPLAB-C18 C Compiler 423
Table 26: Global Salvo Object Codes for Salvo Libraries – Microchip MPLAB-C18 C Compiler 423
Table 27: Processor Types for Salvo Libraries – Microchip MPLAB-C18 C Compiler 424
Table 28: Configuration Options Used to Build Salvo Libraries – HI-TECH 8051C C Compiler 426
Table 29: Memory Model Codes for Salvo Libraries – HI-TECH 8051C C Compiler 427
Table 30: Salvo Object Memory Type Codes for Salvo Libraries – HI-TECH 8051C C compiler 427
Table 31: Configuration Options Used to Build Salvo Libraries – Keil Cx51 C Compiler 429
Table 32: Memory Model Codes for Salvo Libraries – Keil Cx51 C Compiler 430
Table 33: Salvo Object Memory Type Codes for Salvo Libraries – Keil Cx51 C Compiler 430
Table 34: Configuration Options Used to Build Salvo Libraries – ImageCraft ICC11 C compiler 432
Table 35: Configuration Options Used to Build Salvo Libraries – Archelon AQ430 C compiler 435
Table 36: Configuration Options Used to Build Salvo Libraries – IAR MSP430 C compiler 438

Table 37: Configuration Options Used to Build Salvo Libraries – ImageCraft ICC430 C compiler 441
Table 38: Configuration Options Used to Build Salvo Libraries – HI-TECH V8C C Compiler. 444
Table 39: Configuration Options Used to Build Salvo Libraries – GNU C Compiler 447
Table 40: Configuration Options Used to Build Salvo Library – Metrowerks CodeWarrior C Compiler 450
Table 41: Test System Overview 456
Table 42: Features Enabled in Test Configurations I-V 457
Table 43: ROM and RAM Usage for Test Programs 1-5 in Test Systems A & B 458
Table 44: Context-Switching Rates & Times for Test Programs 6-10 in Test Systems A-C 459
Table 45: RAM Requirements for Configurations I-V in Test Systems A-C 462
Table 46: OS_Delay() Execution Times 464
Table 47: OS_Destroy() Execution Times 464
Table 48: OS_Prio() Execution Times 464
Table 49: OS_Stop() Execution Times 464
Table 50: OS_WaitBinSem() Execution Times 464
Table 51: OS_WaitMsg() Execution Times 465
Table 52: OS_WaitMsgQ() Execution Times 465
Table 53: OS_WaitSem() Execution Times 465
Table 54: OS_Yield() Execution Times 465
Table 55: OSCreateBinSem() Execution Times 466
Table 56: OSCreateMsg() Execution Times 466
Table 57: OSCreateMsgQ() Execution Times 466
Table 58: OSCreateSem() Execution Times 466
Table 59: OSCreateTask() Execution Times 466
Table 60: OSInit() Execution Times 467
Table 61: OSSched() Execution Times 467
Table 62: OSSignalBinSem() Execution Times 468
Table 63: OSSignalMsg() Execution Times 468
Table 64: OSSignalMsgQ() Execution Times 468
Table 65: OSSignalSem() Execution Times 469
Table 66: OSStartTask Execution Times 469
Table 67: OSTimer() Execution Times 469
Table 68: Maximum t_InsPrioQ for 1-8 Tasks in Configurations I-V (simple queues) 470
Table 69: Maximum t_DelPrioQ for 1-8 Tasks in Configurations I-V (simple queues) 470
Table 70: Maximum t_InsDelayQ for 1-8 Tasks in Configurations I - V (simple queues, 8-bit delays, w/OSSPEEDUP_QUEUEING) 471
Table 71: Maximum t_InsDelayQ for 1-8 Tasks in Configurations I - V (simple queues, 16-bit delays, w/OSSPEEDUP_QUEUEING) 471
Table 72: Maximum t_DelDelayQ for 1-8 Tasks in Configurations I - V (simple queues, 8-bit delays) 472
Table 73 Maximum t_DelDelayQ for 1-8 Tasks in Configurations I - V (simple queues, 16-bit delays) 472
Table 74: Example of Queueing Operation Times 473
Table 75: t_InsPrioQ for Configurations I & III 475
Table 76: t_InsPrioQ for Configurations II & IV 475
Table 77: t_InsPrioQ for Configuration V 476
Table 78: t_DelPrioQ for Configurations I & III 476
Table 79: t_DelPrioQ for Configurations II & IV 476
Table 80: t_DelPrioQ for Configuration V 476
Table 81: t_InsDelayQ for Configurations II & IV and 8-bit delays 477

Table 82: : t_InsDelayQ for Configurations II & IV and 16-bit delays 477
Table 83: t_InsDelayQ for Configurations II & IV and 8-bit delays, using OSSPEEDUP_QUEUEING 477
Table 84: t_InsDelayQ for Configurations II & IV and 16-bit delays, using OSSPEEDUP_QUEUEING 477
Table 85: t_InsDelayQ for Configuration V and 8-bit delays 477
Table 86: t_InsDelayQ for Configuration V and 16-bit delays 478
Table 87: t_InsDelayQ for Configuration V and 8-bit delays, using OSSPEEDUP_QUEUEING 478
Table 88: t_InsDelayQ for Configuration V and 16-bit delays, using OSSPEEDUP_QUEUEING 478
Table 89: t_DelDelayQ for Configurations II & IV and 8-bit delays 478
Table 90: t_DelDelayQ for Configurations II & IV and 16-bit delays 479
Table 91: t_DelDelayQ for Configuration V and 8-bit delays 479
Table 92: t_DelDelayQ for Configuration V and 16-bit delays 479
Table 93: t_InitTcb for Configuration I 480
Table 94: t_InitTcb for Configuration II 480
Table 95: t_InitTcb for Configuration III 480
Table 96: t_InitTcb for Configuration III 480
Table 97: t_InitTcb for Configuration V 480
Table 98: t_InitEcb for Configuration I 481
Table 99: t_InitEcb for Configuration II 481
Table 100: t_InitEcb for Configuration III 481
Table 101: t_InitEcb for Configuration IV 481
Table 102: t_InitEcb for Configuration V 481
Table 103: Test System Names, Targets and Development Environments 525
Table 104: Configurations for Test Programs t40-t47 540

Release Notes

Introduction

Please refer to the general and distribution-specific release notes that are part of every Salvo distribution for more information on the v3.1.0 release.

Third-Party Tool Versions

Please refer to the distribution-specific release notes for the version numbers of third-party tools (compilers, linkers, librarians, etc.) used to create the Salvo v3.1.0 distributions.

Supported Targets and Compilers

As of v3.1.0, Salvo supports the following targets and compilers:

target	compiler(s)
Intel 8051 family and its derivatives	• HI-TECH 8051C • Keil Cx51
Intel 80x86 family and its derivatives	• gcc (GNU C Compiler) (Cygwin special) • Metrowerks CodeWarrior IDE
Microchip PIC12, PIC1400, PIC16 and PIC17 PICmicro families	• HI-TECH PICC
Microchip PIC18 family	• HI-TECH PICC-18 • IAR PIC18 C • Microchip MPLAB-C18
Motorola M68HC11	• ImageCraft ICC11
TI's MSP430	• Archelon / Quadravox AQ430 • IAR MSP430 C • ImageCraft ICC430
VAutomation V8-µRISC	• HI-TECH V8C

Table 1: Supported Targets and Compilers

Please see Salvo release notes for more information on the version number(s) of the compiler(s) and associated tools (e.g. librarians) used to create each particular distribution. If you have a named compiler that is older than the ones listed, you may need to upgrade it to work with Salvo. Contact the compiler vendor for upgrade information.

Preface

Salvo 2.0 was the first commercial release of Pumpkin, Inc.'s co-operative priority-based multitasking RTOS. Salvo 1.0 was an internal release, written in assembly language and targeted specifically for the Microchip PIC17C756 PICmicro in a proprietary, in-house data acquisition system.

Salvo 1.0 provided most of the basic functionality of 2.0. It was decided to expand on that functionality by rewriting Salvo in C. In doing so, opportunities arose for many configuration options and optimizations, to the point where not only is Salvo more powerful and flexible than its 1.0 predecessor, but it is completely portable, too.

Typographic Conventions

Various text styles are used throughout this manual to improve legibility. Code examples, code excerpts, path names and file names are shown in a `monospaced font`. New and particularly useful terms, and terms requiring emphasis, are shown *italicized*. User input (e.g. at the DOS command line) is shown in this manner. Certain terms and sequence numbers are shown in **bold**. Important notes, cautions and warnings have distinct borders around them:

Note Salvo source code uses tab settings of 4, i.e. tabs are equivalent to 4 spaces.

The letters xyz are used to denote one of several possible names, e.g. `OSSignalXyz()` refers to `OSSignalBinSem()`, `OSSignalMsg()`, `OSSignalMsgQ()`, `OSSignalSem()`, etc. Xyz is case-insensitive.

The symbol | is used as a shorthand to denote multiple, similar names, e.g. `sysa|e|f` denotes `sysa` and/or `syse` and/or `sysf`.

DOS and Windows pathnames use '\'. Linux and Unix pathnames use '/'. They are used interchangeably throughout this document.

Standardized Numbering Scheme

Salvo employs a *standardized numbering scheme* for all software releases. The version/revision numbering scheme uses multiple fields[1] as shown below:

```
salvo-[version-]code-MAJOR.MINOR.SUBMINOR[-PATCH]
```

where

`version` (if present) refers to Salvo Lite
`code` refers to the compiler and target processor(s) supported in the distribution
`MAJOR` changes when major features (e.g. array mode) are added.
`MINOR` changes when minor features (e.g. new user services) are added to or changed.
`SUBMINOR` changes during alpha and beta testing and when support files (e.g. new Application Notes) are added.
`PATCH` is present and changes each time a bug fix is applied and/or new documentation is added.

All `MAJOR.MINOR.SUBMINOR` versions are released with their own, complete installer. `-PATCH` is used only on installers that add new or modified files to an existing Salvo code and documentation installation.

For example, the revisions of Salvo's freeware and full versions might progress like this:

`salvo-lite-pic-2.2.0`	final 2.2 freeware installer for PICmicros, released
`salvo-pic-2.2.0`	final 2.2 installer for PICmicros, released
`salvo-gccx86-2.2.0-3`	2.2 patch installer for gcc compiler and x86 targets with updated documentation and bug fixes
`salvo-gccx86-2.2.1`	`setup-2.2.0` and

[1] The final field is present only on patches.

	setup-2.2.0-3 rolled into a single installer for gcc compiler and x86 targets
salvo-v8c-3.0.alpha1	initial in-house v3.0 installer for V8

Salvo releases are generically referred to by their MAJOR.MINOR numbering, i.e. "the 3.0 release."

The Salvo Coding Mindset

Configurability Is King

Salvo is extremely configurable to meet the requirements of the widest possible target audience of embedded microcontrollers. It also provides you, the user, with all the necessary header files, user hooks, predefined constants, data types, useable functions, etc. that will enable you to create your own Salvo application as quickly and as error-free as possible.

Conserve Precious Resources

The Salvo source code is written first and foremost to use as few resources as possible in the target application. Resources include RAM, ROM, stack call…return levels and instruction cycles. *Most of Salvo's RAM- and ROM-hungry functionality is disabled by default.* If you want a particular feature (e.g. event flags), you must enable it via a *configuration option* (e.g. OSENABLE_EVENT_FLAGS) and re-make your application. This allows you to manage the Salvo code in your application from a single point – the Salvo configuration file salvocfg.h.

Learn to Love the Preprocessor

Salvo makes heavy use of the C preprocessor and symbols predefined by the compiler, Salvo and/or the user in order to configure the source code for compilation. Though this may seem somewhat

daunting at first, you'll find that it makes managing Salvo projects much simpler.

Document, But Don't Duplicate

Wherever possible, neither source code nor documentation is repeated in Salvo. This makes it easier for us to maintain and test the code, and provide accurate and up-to-date information.

We're Not Perfect

While every effort has been made to ensure that Salvo works as advertised and without error, it's entirely possible that we may have overlooked a problem or failed to catch a mistake. Should you find what you think is an error or ambiguity, please contact us so that we can resolve the issue(s) as quickly as possible and enable you to continue coding your Salvo applications worry-free.[2]

Note We feel that it should not be necessary for you to modify the source code to achieve functionality close to what Salvo already provides. We urge you to contact us first with your questions before modifying the source code, as we cannot support modified versions of Salvo. In many instances, we can both propose a solution to your problem, and perhaps also incorporate it into the next Salvo release.

[2] See Pumpkin Salvo Software License Agreement for more information.

Chapter 1 • Introduction

Welcome

In the race to innovate, time-to-market is crucial in launching a successful new product. If you don't take advantage of in-house or commercially available software foundations and support tools, your competition will. But cost is also an important issue, and with silicon (as in real life) prices go up as things get bigger. If your design can afford lots memory and maybe a big microprocessor, too, go out and get those tools. That's what everybody else is doing …

But what if it can't?

What if you've been asked to do the impossible – fit complex, real-time functionality into a low-cost microcontroller and do it all on a tight schedule? What if your processor has only a few KB of ROM and even less RAM? What if the only tools you have are a compiler, some debugging equipment, a couple of books and your imagination? Are you really going to be stuck again with state machines, jump tables, complex interrupt schemes and code that you can't explain to anyone else? After a while, that won't be much fun anymore. Why should you be shut out of using the very same software frameworks the big guys use?

They say that true multitasking needs plenty of memory, and it's not an option for your design. But is that really true?

Not any more. Not with Salvo. Salvo is full-blown multitasking in a surprisingly small memory space – it's about as big as `printf()`![3] Multitasking, priorities, events, a system timer – it's all in there. No stack? That's probably not a problem, either. You'll get more functionality out of your processor quicker than you ever thought possible. And you can put Salvo to work for you right away.

[3] Comparison based on implementations with full `printf()` functionality.

What Is Salvo?

Salvo is a powerful, high-performance and inexpensive real-time operating system (RTOS) that requires very little memory and no stack. It is an easy-to-use software tool to help you quickly create powerful, reliable and sophisticated applications (programs) for embedded systems.

Salvo was designed from the ground up for use in microprocessors and microcontrollers with severely limited resources, and will typically require from 5 to 100 times less memory than other RTOSes. In fact, Salvo's memory requirements are so minimal that it will run where no other RTOS can.

Salvo is ROMable, easily scaleable and extremely portable. It runs on just about any processor, from a PIC to a Pentium.

Why Should I Use Salvo?

If you're designing the next hot embedded product, you know that time-to-market is crucial to guarantee success. Salvo provides a powerful and flexible framework upon which you can quickly build your application.

If you're faced with a complex design and limited processing resources, Salvo can help you make the most of what's available in your system.

And if you're trying to cost-reduce or add functionality to an existing design, Salvo may be what you need because it helps you leverage the processing power you already have.

Before Salvo, embedded systems programmers could only dream of running an RTOS in their low-end processors. They were locked out of the benefits that an RTOS can bring to a project, including reducing time-to-market, managing complexity, enhancing robustness and improving code sharing and re-use. They were unable to take advantage of the many well-established RTOS features designed to solve common and recurring problems in embedded systems programming.

That dream is now a reality. With Salvo, you can stop worrying about the underlying structure and reliability of your program and start focusing on the application itself.

What Kind of RTOS Is Salvo?

Salvo is a cooperative multitasking RTOS, with full support for event and timer services. Multitasking is priority-based, with fifteen separate priority levels supported. Tasks that share the same priority will execute in a round-robin fashion. Salvo provides services for employing semaphores, messages and message queues for intertask communications and resource management. A full complement of RTOS functions (e.g. context-switch, stop a task, wait on a semaphore, etc.) is supported. Timer functions, including delays and timeouts, are also supported.

Salvo is written in ANSI C, with a very small number of processor-specific extensions, some of which are written in native assembly language. It is highly configurable to support the unique demands of your particular application.

While Salvo is targeted towards embedded applications, it is universally applicable and can also be used to create applications for other types of systems (e.g. 16-bit DOS applications).

What Does a Salvo Program Look Like?

A Salvo program looks a lot like any other that runs under a multitasking RTOS. Listing 1 shows (with comments) the source code for a remote automotive seat warmer with user-settable temperature. The microcontroller is integrated into the seat, and requires just four wires for communication with the rest of the car's electronics – power, ground, Rx (to receive the desired seat temperature from a control mounted elsewhere) and Tx (to indicate status). The desired temperature is maintained via `TaskControl()`. `TaskStatus()` sends, every second, either a single 50ms pulse to indicate that the seat has not yet warmed up, or two consecutive 50ms pulses to indicate that the seat is at the desired temperature.

```
#include <salvo.h>

typedef unsigned char t_boolean;
typedef unsigned char t_temp;

/* Salvo context-switching labels              */
_OSLabel(TaskControl1)
_OSLabel(TaskStatus1)
_OSLabel(TaskStatus2)
_OSLabel(TaskStatus3)
_OSLabel(TaskStatus4)

/* local flag                                  */
```

```
t_boolean warm = FALSE;

/* seat temperature functions                  */
extern t_temp UserTemp( void );
extern t_temp SeatTemp( void );
extern t_boolean CtrlTemp( t_temp user, seat );

/* initialize Salvo, create and assign         */
/*  priorities to the tasks, and begin         */
/*  multitasking.                              */
int main( void )
{
  OSInit();

  OSCreateTask(TaskControl, OSTCBP(1), 8);
  OSCreateTask(TaskStatus,  OSTCBP(2), 3);

  for (;;)
    OSSched();
}

/* moderate-priority (i.e. 8) task (i.e. #1)   */
/*  to maintain seat temperature.  CtrlTemp()  */
/*  returns TRUE only if the seat is at the    */
/*  the desired (user) temperature.            */
void TaskControl( void )
{
  for (;;) {
    warm = CtrlTemp(UserTemp(), SeatTemp());
    OS_Yield(TaskControl1);
  }
}

/* high-priority (i.e. 3) task (i.e. #2) to    */
/*  generate pulses. System ticks are 10ms.    */
void TaskStatus( void )
{
  /* initialize pulse output (low).            */
  TX_PORT &= ~0x01;

  for (;;) {
    OS_Delay(100, TaskStatus1);
    TX_PORT |= 0x01;
    OS_Delay(5, TaskStatus2);
    TX_PORT &= ~0x01;
    if (warm) {
      OS_Delay(5, TaskStatus3);
      TX_PORT |= 0x01;
      OS_Delay(5, TaskStatus4);
      TX_PORT &= ~0x01;
    }
  }
}
```

Listing 1: A Simple Salvo Program

It's important to note that when this program runs, temperature control continues while `TaskStatus()` is delayed. The calls to `OS_Delay()` do not cause the program to loop for some amount of time and then continue. After all, that would be a waste of processor resources (i.e. instruction cycles). Instead, those calls simply instruct Salvo to suspend the pulse generator and ensure that it resumes running after the specified time period. `TaskControl()` runs whenever `TaskStatus()` is suspended.

Apart from creating a simple Salvo configuration file and tying Salvo's timer to a 10ms periodic interrupt in your system, the C code above is all that is needed to run these two tasks concurrently. Imagine how easy it is to add more tasks to this application to enhance its functionality.

See *Chapter 4 • Tutorial* for more information on programming with Salvo.

What Resources Does Salvo Require?

Salvo neither uses nor requires a general-purpose stack. This means that even if your processor does not have PUSH and POP instructions, or stack registers, you can probably use Salvo. The only stack that Salvo requires is one that supports function calls and returns, i.e. a so-called call ... return or hardware stack.

The amount of ROM Salvo requires will depend on how much of Salvo you are using. A minimal multitasking application on a RISC processor might use a few hundred instructions. A full-blown Salvo application on the same processor will use around 1K instructions.

The amount of RAM Salvo requires is also dependent on your particular configuration. In a RISC application,[4] each task will require 4-12 (typically 7) bytes, each event 3-4 bytes,[5] and 4-6 more bytes are required to manage all the tasks, events and delays. That's it!

In all cases, the amount of RAM required is primarily dependent on the size of pointers (i.e. 8 or 16 bits) to ROM and RAM in your application, i.e. it's application-dependent. In some applications (e.g. CISC processors) additional RAM may be required for general-purpose register storage.

---

4 PIC16 series (e.g. PIC16C64). Pointers to ROM take two bytes, and pointers to RAM take one byte.

5 Message queues require additional RAM.

If you plan to use the delay and timeout services, Salvo requires that you provide it with a single interrupt. This interrupt need not be dedicated to Salvo – it can be used for your own purposes, too.

The number of tasks and events is limited only by the amount of available memory.

See *Chapter 6 • Frequently Asked Questions (FAQ)* for more information.

How Is Salvo Different?

Salvo is a cooperative RTOS that doesn't use a stack.[6] Virtually all other RTOSes use a stack, and many are preemptive as well as cooperative. This means that compared to other RTOSes, Salvo differs primarily in these ways:

- Salvo is a cooperative RTOS, so you must explicitly manage task switching[7].
- Task switching can only occur at the task level, i.e. directly inside your tasks, and not from within a function called by your task, or elsewhere. This is due to the absence of a general-purpose stack, and may have a small impact on the structure of your program.
- Compared to other cooperative or preemptive RTOSes, which need lots of RAM memory (usually in the form of a general-purpose stack), Salvo needs very little. For processors without much RAM, Salvo may be your only RTOS choice.

Salvo is able to provide most of the performance and features of a full-blown RTOS while using only a fraction as much memory. With Salvo you can quickly create powerful, fast, sophisticated and robust multitasking applications.

---

6 By "stack" we mean a general-purpose stack that can be manipulated (e.g. via PUSH and POP instructions) by the programmer. Salvo still requires a call ... return stack, sometimes called a "hardware stack."

7 We'll explain this term later, but for now it means being in one task and relinquishing control of the processor so that another task may run.

What Do I Need to Use Salvo?

A working knowledge of C is recommended. But even if you're a C beginner, you shouldn't have much difficulty learning to use Salvo.

Some knowledge of RTOS fundamentals is useful, but not required. If working with an RTOS is new to you, be sure to review *Chapter 2 • RTOS Fundamentals*.

You will need a good ANSI-C-compliant compiler for the processor(s) you're using. It must be capable of compiling the Salvo source code, which makes use of many C features, including (but not limited to):

- arrays,
- unions,
- bit fields,
- structures,
- static variables,
- multiple source files,
- indirect function calls,
- multiple levels of indirection,
- passing of all types of parameters,
- multiple bytes of parameter passing,
- extensive use of the C preprocessor and
- pointers to functions, arrays, structures, unions, etc.
- support for variable arguments lists[8] (via `va_arg()`, etc.)

Listing 2: C Compiler Feature Requirements

Your compiler should also be able to perform in-line assembly. The more fully-featured the in-line assembler, the better.

Lastly, your compiler should be capable of compiling to object (`*.o`) modules and libraries (`*.lib`), and linking object modules and libraries together to form a final executable (usually `*.hex`).

We recommend that you use a compiler that is already certified for use with Salvo. If your favorite compiler and/or processor are not yet supported, you can probably do a port to them in a few hours. *Chapter 10 • Porting* will guide you through the process. Always check with the factory for the latest news concerning supported compilers and processors.

[8] This is not absolutely necessary, but is desireable. `va_arg()` is part of the ANSI C standard.

Which compilers and Processors does Salvo support?

- HI-TECH PICC: All Microchip PIC12, PIC14000, PIC16 and PIC17 PICmicro MCUs
- HI-TECH PICC Lite: Microchip PIC16C84, PIC16F84 and PIC16F84A MCUs
- HI-TECH PICC-18: All Microchip PIC18 PICmicro MCUs
- Metrowerks CodeWarrior: x86 family
- Mix PowerC: x86 family

How Is Salvo Distributed?

Salvo is supplied on CD-ROM or downloadable over the Internet as a Windows 95 / 98 / ME / NT / 2000 / XP install program. After you install Salvo onto your computer you will have a group of subdirectories that contain the Salvo source code, examples, demo programs, this manual and various other support files.

What Is in this Manual?

Chapter 1 • Introduction is this chapter.

Chapter 2 • RTOS Fundamentals is an introduction to RTOS programming. If you're only familiar with traditional "superloop" or "foreground / background" programming architectures, you should definitely review this chapter.

Chapter 3 • Installation covers how to install Salvo onto your computer.

Chapter 4 • Tutorial is a guide to using Salvo. It contains examples to introduce you to all of Salvo's functionality and how to use it in your application. Even programmers familiar with other RTOSes should still review this chapter.

Chapter 5 • Configuration explains all of Salvo's configuration parameters. Beginners and experienced users need this information to optimize Salvo's size and performance to their particular application.

Chapter 6 • Frequently Asked Questions (FAQ) contains answers to many frequently asked questions.

Chapter 7 • Reference is a guide to all of Salvo's user services (callable functions).

Chapter 8 • Libraries lists the available freeware and standard libraries and explains how to use them.

Chapter 9 • Performance has actual data on the size and speed of Salvo in various configurations. It also has tips on how to characterize Salvo's performance in your particular system.

Chapter 10 • Porting covers the issues you'll face if you're porting Salvo to a compiler and/or processor that is not yet formally certified or supported by Salvo.

Chapter 11 • Tips, Tricks and Troubleshooting has information on a variety of problems you may encounter, and how to solve them.

Appendix A • Recommended Reading contains references to multitasking and related documents.

Appendix B • Other Resources has information on other resources that may be useful to you in conjunction with Salvo.

Appendix C • File and Program Descriptions contains descriptions of all of the files and file types that are part of a Salvo installation.

Chapter 2 • RTOS Fundamentals

Note If you're already familiar with RTOS fundamentals you may want to skip directly to *Chapter 3 • Installation*.

Introduction

> *"I've built polled systems. Yech. Worse are applications that must deal with several different things more or less concurrently, without using multitasking. The software in both situations is invariably a convoluted mess. Twenty years ago, I naïvely built a steel thickness gauge without an RTOS, only to later have to shoehorn one in. Too many asynchronous things were happening; the in-line code grew to outlandish complexity."* Jack G. Ganssle[9]

Most programmers are familiar with traditional systems that employ a looping construct for the main part of the application and use interrupts to handle time-critical events. These are so-called *foreground / background* (or *superloop*) systems, where the interrupts run in the *foreground* (because they take priority over everything else) and the main loop runs in the *background* when no interrupts are active. As applications grow in size and complexity this approach loses its appeal because it becomes increasingly difficult to characterize the interaction between the foreground and background.

An alternative method for structuring applications is to use a software framework that manages overall program execution according to a set of clearly defined rules. With these rules in place, the application's performance can be characterized in a relatively straightforward manner, regardless of its size and complexity.

Many embedded systems can benefit from using an approach involving the use of multiple, concurrent tasks communicating amongst themselves, all managed by a kernel, and with clearly-defined run-time behavior. This is the RTOS approach to programming. These and other terms are defined below.

---

9 "Interrupt Latency", *Embedded Systems Programming*, Vol. 14 No. 11, October 2001, p. 73.

Note This chapter is only a quick introduction to the operation and use of an RTOS. *Appendix A • Recommended Reading* contains references for further, in-depth reading.

Basic Terms

A *task* is a sequence of instructions, sometimes done repetitively, to perform an action (e.g. read a keypad, display a message on an LCD, flash an LED or generate a waveform). In other words, it's usually a small program inside a bigger one. When running on a relatively simple processor (e.g. Z80, 68HC11, PIC), a task may have all of the system's resources to itself regardless of how many tasks are used in the application.

An *interrupt* is an internal or external hardware event that causes program execution to be suspended. Interrupts must be enabled for an interrupt to occur. When this occurs, the processor vectors to a user-defined *interrupt service routine (ISR)*, which runs to completion. Then program execution picks up where it left off. Because of their ability to suspend program execution, interrupts are said to run in the foreground, and the rest of the program runs in the background.

A task's *priority* suggests the task's importance relative to other tasks. It may be fixed or variable, unique or shared with other tasks.

A *task switch* occurs when one task suspends running and another starts or resumes running. It may also be called a *context switch*, because a task's context (generally the complete contents of the stack and the values of the registers) is usually saved for re-use when the task resumes.

Preemption occurs when a task is interrupted and another task is made ready to run. An alternative to a preemptive system is a *cooperative* system, in which a task must voluntarily relinquish control of the processor before another task may run. It is up to the programmer to structure the task so that this occurs. If a running task fails to cooperate, then no other tasks will execute, and the application will fail to work properly.

Preemptive and cooperative context switching are handled by a *kernel*. Kernel software manages the switching of tasks (also called *scheduling*) and intertask communication. A kernel generally ensures that the highest-priority eligible task is the task that's running

(preemptive scheduling) or will run next (cooperative scheduling). Kernels are written to be as small and as fast as possible to guarantee high performance in the overlying application program.[10]

A *delay* is an amount of time (often specified in milliseconds) during which a task's execution can be suspended. While suspended, a task should use as few of the processor's resources as possible to maximize the performance of the overall application, which is likely to include other tasks that are not concurrently suspended. Once the delay has *elapsed* (or *expired*), the task resumes executing. The programmer specifies how long the delay is, and how often it occurs.

An *event* is an occurrence of something (e.g. a key was pressed, an error occurred or an expected response failed to occur) that a task can wait for. Also, just about any part of a program can signal the occurrence of an event, thus letting others know that the event happened.

Intertask communication is an orderly means of passing information from one task to another following some well-established programming concepts. *Semaphores*, *messages*, *message queues* and *event flags* can be used to pass information in one form or another between tasks and, in some cases, ISRs.

A *timeout* is an amount of time (often specified in milliseconds) that a task can wait for an event. Timeouts are optional – a task can also wait for an event indefinitely. If a task specifies a timeout when waiting for an event and the event doesn't occur, we say that a timeout has occurred, and special handling is invoked.

A task's *state* describes what the task is currently doing. Tasks change from one state to another via clearly defined rules. Common task states might be *ready / eligible*, *running*, *delayed*, *waiting*, *stopped* and *destroyed / uninitialized*.

The *timer* is another piece of software that keeps track of elapsed time and/or real time for delays, timeouts and other time-related services. The timer is only as accurate as the timer clock provided by your system.

A system is *idling* when there are no tasks to run.

---

[10] Some kernels also provide I/O functions and other services such as memory management. Those are not discussed here.

The *operating system (OS)* contains the kernel, the timer and the remaining software (called *services*) to handle tasks and events (e.g. task creation, signaling of an event). One chooses a *real-time operating system (RTOS)* when certain operations are critical and must be completed correctly and within a certain amount of time. An RTOS-enabled *application* or *program* is the end product of combining your tasks, ISRs, data structures, etc, with an RTOS to form single program.

Now let's examine all these terms, and some others, in more detail.

Foreground / Background Systems

The simplest program structure is one of a *main loop* (sometimes called a *superloop*) calling functions in an ordered sequence. Because program execution can switch from the main loop to an ISR and back, the main loop is said to run in the background, whereas the ISRs run in the foreground. This is the sort of programming that many beginners encounter when learning to program simple systems.

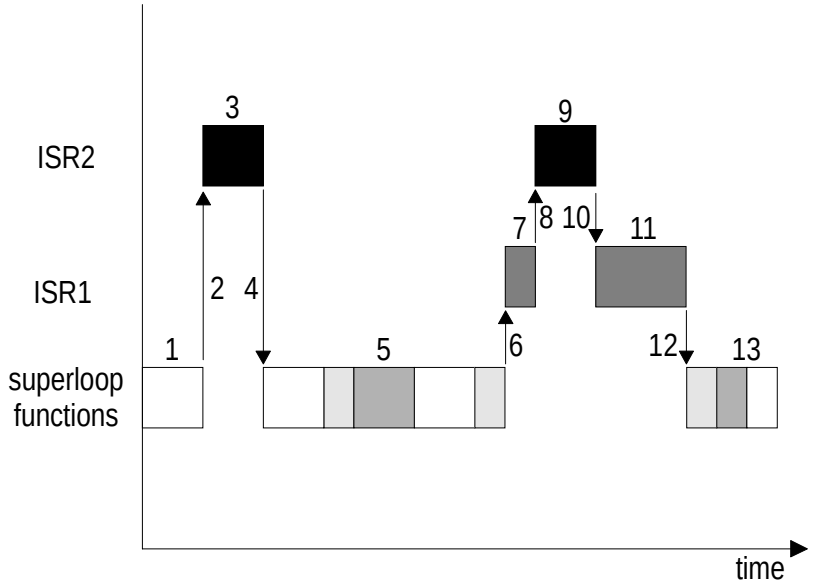


Figure 1: Foreground / Background Processing

In Figure 1 we see a group of functions repeated over and over [1, 5, 13] in a main loop. Interrupts may occur at any time, and even at multiple levels. When an interrupt occurs (high-priority interrupt at [2] and [8], low-priority interrupt at [6]), processing in the function is suspended until the interrupt is finished, whereupon the program returns to the main loop or to a previous interrupted ISR. The main loop functions are executed in strictly serial manner, all at the same

priority, without any means of changing when or even if the function should execute. ISRs must be used in order to respond quickly to external events, and can be prioritized if multiple interrupt levels are supported.

Foreground / background systems are relatively simple from a programming standpoint as long as there is little interaction amongst the functions in the main loop and between them and the ISRs. But they have several drawbacks: Loop timing is affected by any changes in the loop and/or ISR code. Also, the *response* of the system to inputs is poor because information made available by an ISR to a function in the loop cannot be processed by the function until its turn to execute. This rigidly sequential nature of program execution in the super loop affords very little flexibility to the programmer, and complicates time-critical operations. State machines may be used to partially solve this problem. As the application grows, the loop timing becomes unpredictable, and a variety of other complicating factors arise.

Reentrancy

One such factor is *reentrancy*. A reentrant function can be used simultaneously in one or more parts of an application without corrupting data. If the function is not written to be reentrant, simultaneous calls may corrupt the function's internal data, with unpredictable results in the application. For example, if an application has a non-reentrant `printf()` function and it is called both from main loop code (i.e. the background) and also from within an ISR (i.e. the foreground), there's an excellent chance that every once in a while the resultant output of a call to

```
printf("Here we are in the main loop.\n");
```

from within the main loop and a call to

```
printf("Now we are servicing an interrupt.\n");
```

from within an ISR at the same time might be

```
Here we aNow we are servicing an interrupt.
```

Listing 3: Reentrancy Errors with printf()

This is clearly in error. What has happened is that the first instance of `printf()` (called from within the main loop) got as far as printing the first 9 characters ("`Here we a`") of its string argument before being interrupted. The ISR also included a call to `printf()`,

which re-initialized its local variables and succeeded in printing its entire 36-character string ("`Now we … interrupt.\n`"). After the ISR finished, the main-loop `printf()` resumed where it had left off, but its internal variables reflected having successfully written to the end of a string argument, and no further output appeared necessary, so it simply returned and the main loop continued executing.

Note Calling non-reentrant functions as if they were reentrant rarely results in such benign behavior.

Various techniques can be employed to avoid this problem of a non-reentrant `printf()`. One is to disable interrupts before calling a non-reentrant function and to re-enable them thereafter. Another is to rewrite `printf()` to only use local variables (i.e. variables that are kept on the function's stack). The stack plays a very important role in reentrant functions.

Resources

A *resource* is something within your program that can be used by other parts of the program. A resource might be a register, a variable or a data structure, or it might be something physical like an LCD or a beeper. A *shared resource* is a resource that may be used by more than one part of your program. If two separate parts of a program are contending for the same resource, you'll need to manage this by *mutual exclusion*. Whenever a part of your program wants to use the resource it must obtain exclusive access to it in order to avoid corrupting it.

Multitasking and Context Switching

Many advantages can be realized by splitting a foreground / background application into one with multiple, independent tasks. In order to *multitask*, such that all tasks appear to run concurrently, some mechanism must exist to pass control of the processor and its resources from one task to another. This is the job of the *scheduler*, part of the kernel that (among its other duties) suspends one task and resumes another when certain conditions are met. It does this by storing the program counter for one task and restoring the program counter for another. The faster the scheduler is able to switch tasks, the better the performance of the overall application, since the time spent switching tasks is time spent without any tasks running.

A context switch must appear transparent to the task itself. The task's "world view" before the context switch that suspends it and after the context switch that resumes it must be the same. This way, task A can be interrupted at any time to allow the scheduler to run a higher-priority task, task B. Once task B is finished, task A resumes where it left off. The only effect of the context switch on task A is that it was suspended for a potentially long time as a result of the context switch. Hence tasks that have time-critical operations must prevent context switches from occurring during those critical periods.

From a task's perspective, a context switch can be "out of the blue", in the sense that the context switch was forced upon it for reasons external to the task, or it can be intentional due to the programmer's desire to temporarily suspend the task to do other things.

Most processors support general-purpose stacks and have multiple registers. Just restoring the appropriate program counter will not be enough to guarantee the continuity of a task's execution. That's because the stack and the register values will be unique to that task at the moment of the context switch. A context switch saves the entire task's context (e.g. program counter, registers, stack contents). Most processor architectures require that memory must be allocated to each task to support context switching.

Tasks and Interrupts

As is the case with foreground / background systems, multitasking systems often make extensive use of interrupts. Tasks must be protected from the effects of interrupts, ISRs should be as fast as possible, and interrupts should be enabled most of the time. Interrupts and tasks coexist simultaneously – an interrupt may occur right in the middle of a task. The disabling of interrupts during a task should be minimized, yet interrupts will have to be controlled to avoid conflicts between tasks and interrupts when shared resources are accessed by both.

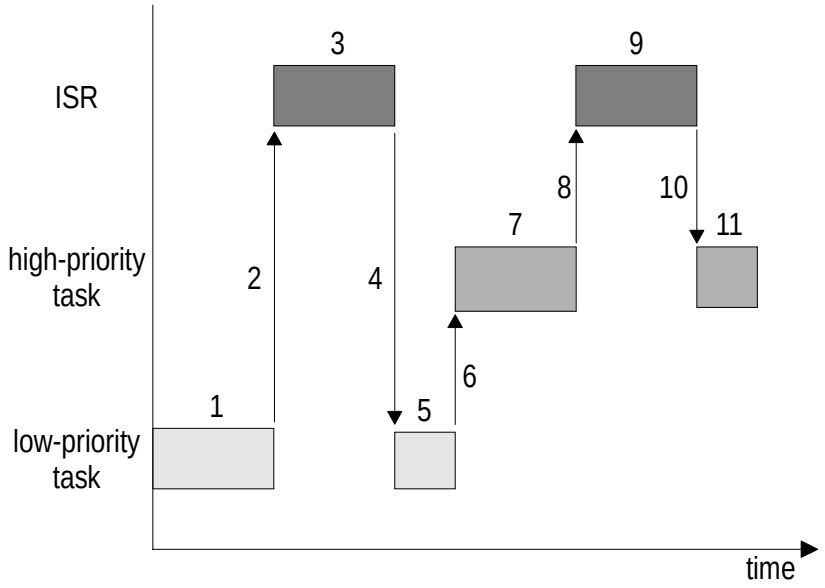


Figure 2: Interrupts Can Occur While Tasks Are Running

In Figure 2 a low-priority task is running [1] when an interrupt occurs [2]. In this example, interrupts are always enabled. The interrupt [3] runs to completion [4], whereupon the low-priority task [5] resumes its execution. A context switch occurs [6] and the high-priority task [7] begins executing. The context switch is handled by the scheduler (not shown). The high-priority task is also interrupted [8-10] before continuing [11].

Interrupt latency is defined as the maximum amount of time that interrupts are disabled, plus the time it takes to execute the first instruction of an ISR. In other words, it's the worst-case delay between when an interrupt occurs and when the corresponding ISR begins to execute.

Preemptive vs. Cooperative Scheduling

There are two types of schedulers: preemptive and cooperative. A *preemptive scheduler* can cause the current task (i.e. the task that's currently running) to be preempted by another one. Preemption occurs when a task with higher priority than the current task becomes eligible to run. Because it can occur at any time, preemption requires the use of interrupts and stack management to guarantee the correctness of the context switch. By temporarily disabling preemption, programmers can prevent unwanted disruptions in their programs during critical sections of code.

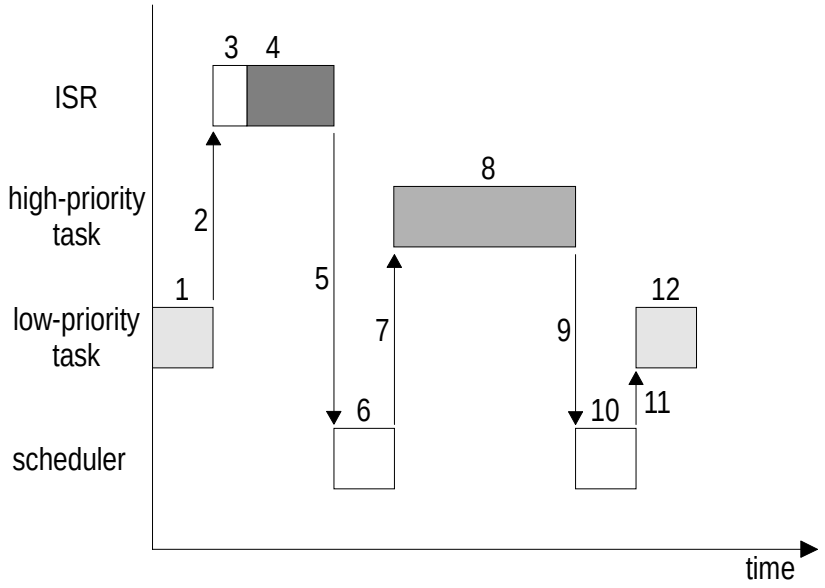


Figure 3: Preemptive Scheduling

Preemptive Scheduling

Figure 3 illustrates the workings of a preemptive scheduler. A low-priority task [1] is running when an external event occurs [2] that triggers an interrupt. The task's context and some other information for the scheduler are first saved [3] in the ISR, and the interrupt is serviced [4]. In this example the high-priority task is waiting for this particular event and should run as soon as possible after the event occurs. When the ISR is finished [5], it proceeds to the scheduler [6], which starts [7] the high-priority task [8]. When it is finished, control returns to the scheduler [9, 10], which then restores the low-priority task's context and allows it to resume where it was interrupted [11, 12].

Preemptive scheduling is very stack-intensive. The scheduler maintains a separate stack for each task so that when a task resumes execution after a context switch, all the stack values that are unique to the task are properly in place. These would normally be return addresses from subroutine calls, and parameters and local variables (for a language like C). The scheduler may also save a suspended task's context on the stack, since it may be convenient to do so.

Preemptive schedulers are generally quite complex because of the myriad of issues that must be addressed to properly support context switching at any time. This is especially true with regard to the handling of interrupts. Also, as can be seen in Figure 3, a certain

time lag exists between when an interrupt happens and when the corresponding ISR can run. This, plus the interrupt latency, is the *interrupt response* time (t_4 - t_2). The time between the end of the ISR and the resumption of task execution is the *interrupt recovery* time (t_7 – t_5). The system's *event response time* is shown as (t_7 - t_2).

Cooperative Scheduling

A *cooperative scheduler* is likely to be simpler than its preemptive counterpart. Since the tasks must all cooperate for context switching to occur, the scheduler is less dependent on interrupts and can be smaller and potentially faster. Also, the programmer knows exactly when context switches will occur, and can protect critical regions of code simply by keeping a context-switching call out of that part of the code. With their relative simplicity and control over context switching, cooperative schedulers have certain advantages.

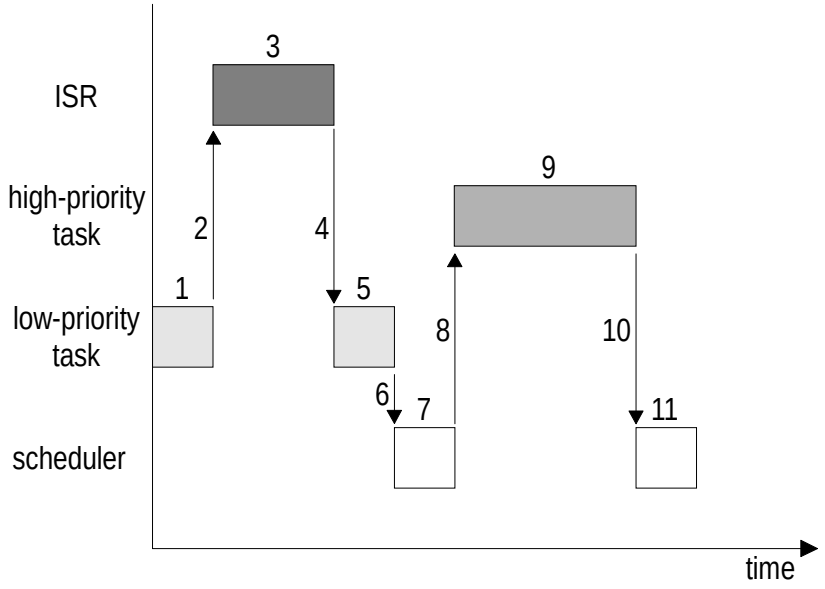


Figure 4: Cooperative Scheduling

Figure 4 illustrates the workings of a cooperative scheduler. As in the previous example, the high-priority task will run after the interrupt-driven event occurs. The event occurs while the low-priority task is running [1, 5]. The ISR is serviced [2-4] and the scheduler is informed of the event, but no context switch occurs until the low-priority task explicitly allows it [6]. Once the scheduler has a chance to run [7], it starts and runs the high-priority task to completion [8-10]. The scheduler [11] will then start whichever eligible task has the highest priority.

In comparison to the preemptive scheduling, cooperative scheduling has the advantage of shorter interrupt response and recovery times and greater overall simplicity. However, the responsiveness is worse because a high-priority eligible task cannot run until a lower-priority one has relinquished control of the processor via an explicit context switch.

More on Multitasking

You can think of tasks as little programs that run within a bigger program (your application). In fact, by using a multitasking RTOS your application can be viewed as a framework to define tasks and to control how and when they run. When your application is running, it means that a bunch of little programs (the tasks) are all running in a manner that makes it appear as if they execute simultaneously. Of course only one task can actually run at a particular instant. In order to take full advantage of the multitasking abilities of the RTOS, you want to define your tasks such that at any particular time, the processor is making the best use of its processing power by running whichever task is most important. Once your task priorities are correctly defined, the scheduler will take care of the rest.

Task Structure

What does a task in a multitasking application actually look like? A task is generally an operation that needs to occur over and over again in your application. The structure is really very simple, and consists of an optional initialization, and then a main loop that is repeated unconditionally. When used with a preemptive scheduler, a task might look like this:

```
Initialize();
for (;;) {
  ...
}
```

Listing 4: Task Structure for Preemptive Multitasking

because a preemptive scheduler can interrupt a task at any time. With a cooperative scheduler a task might look like this:

```
Initialize();
for (;;) {
  ...
  TaskSwitch();
  ...
```

```
}
```

Listing 5: Task Structure for Cooperative Multitasking

The only difference between the two versions is the need to explicitly call out the context switch in the cooperative version. In cooperative multitasking it's up to each task to declare when it is willing to potentially relinquish control of the processor to another task. Such context switches are usually unconditional – a trip through the scheduler may be required even if the current task is the only task eligible to run. In preemptive multitasking this would never occur, as the scheduler would force a context switch only when a higher-priority task had become eligible to run.

Note Context switches can occur multiple times inside a task, both in preemptive and cooperative multitasking systems.

Simple Multitasking

The simplest form of multitasking involves "sharing" the processor equally between two or more tasks. Each task runs, in turn, for some period of time. The tasks *round-robin*, or execute one after the other, indefinitely.

This has limited utility, and suffers from the problems of a super-loop architecture. That's because all tasks have equal, unweighted access to the processor, and their sequence of execution is likely to be fixed.

Priority-based Multitasking

Adding priorities to the tasks changes the situation dramatically. That's because by assigning task priorities you can guarantee that at any instant, your processor is running the most important task in your system.

Priorities can be static or dynamic. *Static priorities* are priorities assigned to tasks at compile time that do not change while the application is running. With *dynamic priorities* a task can change its priority during runtime.

Is should be apparent that if the highest-priority task were allowed to run continuously, then the system would no longer be multitasking. How can multiple tasks with different priorities coexist in a multitasking system? The answer lies in how tasks actually behave – they're not always running! Instead, what a certain task is doing

at any particular time depends on its *state* and on other factors, like *events*.

Task States

An RTOS maintains each task in one of a number of task *states*. Figure 5 illustrates the different states a task can be in, and the allowed transitions between states. *Running* is only one of several exclusive task states. A task can also be *eligible* to run, it can be *delayed*, it can be *stopped* or even *destroyed* / *uninitialized*, and it can be *waiting* for an event. These are explained below.

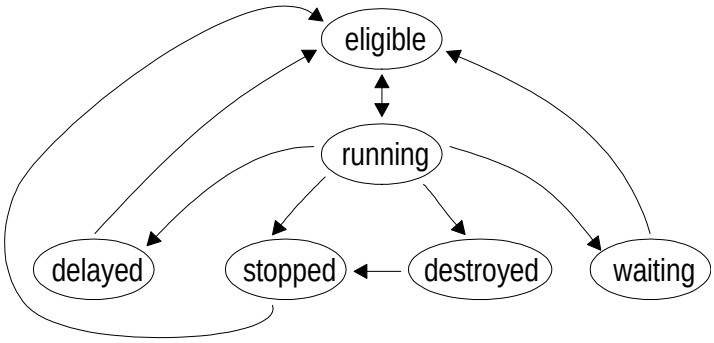


Figure 5: Task States

Before a task is created, it is in the uninitialized state. It returns to that state when and if it is destroyed. There's not much you can do with a destroyed task, other than create another one in its place, or recreate the same task again. A task *transitions* from the destroyed state to the stopped state when it is created via a call to the RTOS *service* that creates a task.

An eligible task is one that is ready to run, but can't because it's not the task with the highest priority. It will remain in this state until the scheduler determines that it is the highest-priority eligible task and makes it run. Stopped, delayed and/or waiting tasks can become eligible via calls to the corresponding RTOS services.

A running task will return to the eligible state after a simple context switch. However, it may transition to a different state if either the task calls an RTOS service that destroys, stops, delays or waits the task, or the task is forced into one of these states via a call to an RTOS service from elsewhere in your application.

A delayed task is one that was previously running but is now suspended and is waiting for a delay timer to expire. Once the timer has expired, the RTOS timer makes the task eligible again.

A stopped task was previously running, and was then suspended indefinitely. It will not run again unless it is (re-)started via a call to the RTOS service that starts a task.

A waiting task is suspended and will remain that way until the event it is waiting for occurs (See "Event-driven Multitasking" below).

It's typical for a multitasking application to have its various tasks in many different states at any particular instant. Periodic tasks are likely to be delayed at any particular instant. Low-priority tasks may be eligible but unable to run because a higher-priority task is already running. Some tasks are likely to be waiting for an event. Tasks may even be destroyed or stopped. It's up to the scheduler to manage all these tasks and guarantee that each tasks runs when it should. The scheduler and other parts of the RTOS ensure that tasks transition from one state to the next properly.

Note The heart of a priority-based multitasking application, the scheduler, is concerned with only one thing – running the highest-priority task that's eligible to run. Generally speaking, the scheduler interacts only with the running task and tasks that are eligible to run.

An RTOS is likely to treat all tasks in a particular state in the same manner, and thereby improve the performance of your application. For example, it shouldn't expend *any* processor cycles on tasks that are stopped or destroyed. After all, they're just "sitting there" and will remain so indefinitely, or until your program makes them eligible to run.

Delays and the Timer

Most embedded programmers are familiar with the simple delay loop construct, e.g.:

```
…
for ( i=0; i<100; i++ )
  asm("nop"); /* do nothing for 100 somethings */
…
```

Listing 6: Delay Loop

The trouble with doing delays like the one shown in Listing 6 is that your application can't do any useful background processing while the loop is running. Sure, interrupts can occur in the foreground, but wouldn't it be nice to be able to do something else during the delay?

Another problem with the code in Listing 6 is that it is compiler-, processor- and speed-dependent. The compiler may or may not optimize the assembly instructions that make up this loop, leading to variations in the actual delay time. Changing the processor may change the delay time, too. And if you increase the processor clock, the delay will decrease accordingly. In order to circumvent these problems delay loops are often coded in assembly language, which severely limits code portability.

An RTOS provides a mechanism for tracking elapsed time through a system timer. This timer is often called in your application via a periodic interrupt. Each time it is called, the timer increments a counter that holds the number of elapsed *system ticks*. The current value of the system ticks is usually readable, and perhaps writeable too, in order to reset it.

The rate at which the timer is called is chosen to yield enough resolution to make it useful for time-based services, e.g. to delay a task or track elapsed time. A fluid level monitor can probably make do with a *system tick rate* of 1Hz (i.e. 1s system ticks), whereas a keypad reader might need a system tick rate of 100Hz (i.e. 10ms system ticks) in order to specify delays for the key debounce algorithm. An unduly fast system tick rate will result in substantial overhead and less processing power left over for your application, and should be avoided.

There must also be enough storage allocated to the system ticks counter to ensure that it will not overflow during the longest time period that you expect to use it. For example, a one-byte timer and a 10ms system tick period will provide a maximum specifiable task delay of 2.55s. In this example, attempting to calculate an elapsed time via the timer will result in erroneous results if successive reads are more than 2.55s apart. Task delays fall under similar restrictions. For example, a system with 10ms system ticks and support for 32-bit delays can delay a task up to a whopping 497 days!

Since the use of delays is common, an RTOS may provide built-in delay services, optimized to keep overhead to a minimum and to boost performance. By putting the desired delay inside a task, we can suspend the task while the delay is counting down, and then

resume the task once the delay has expired. Specifying the delay as a real amount of time will greatly improve our code's portability, too. The code for delaying a task via the RTOS looks quite different than that of Listing 6:

```
…
OS_Delay(100); /* delay for 100 ticks @ 50Hz */
…
```

Listing 7: Delaying via the RTOS

In Listing 7, the call to the RTOS service `OS_Delay()` changes the state of the task from running to delayed. Since the task is no longer running, nor is it even eligible to run (remember, it's delayed), a context switch also occurs, and the highest-priority eligible task (if there is one) starts running.

In Listing 7 `OS_Delay()` also specifies a delay of 100 system ticks. If the system in has a system tick rate of 50Hz, then the task will be delayed for (100 ticks x 20ms) – two full seconds – before resuming execution once it becomes the highest-priority eligible task. Imagine how much processing other eligible tasks can do in two full seconds!

An RTOS can support multiple, simultaneously delayed tasks. It's up to the RTOS designer to maximize performance – i.e. minimize the overhead associated with the execution of the timer – regardless of how many tasks are delayed at any time. This timer overhead cannot be eliminated; it can only be minimized.

The *resolution* and *accuracy* of the system timer may be important to your application. In a simple RTOS, the resolution and the accuracy of the timer both equal the system tick period. For example, delaying a task by `n` system ticks will result in a delay ranging from just over `n-1` to just under `n` system ticks of real time (e.g. milliseconds). This is due to the asynchronous nature of the system timer – if you delay a task immediately after the (interrupt) call to the timer, the first delay tick will last nearly a full system tick. If, on the other hand, you delay a task immediately prior to a system tick, the first delay tick will be very short indeed.

Event-driven Multitasking

You may have noticed that a delayed task is actually waiting for something – it's waiting for its delay timer to expire. The expiration of a delay timer is an example of an *event*, and events may

cause a task to change state. Therefore events are used to control task execution. Examples of events include:

- an interrupt,
- an error occurring,
- a timer timing out,
- a periodic interrupt,
- a resource being freed,
- an I/O pin changing state,
- a key on a keypad being pressed,
- an RS-232 character being received or transmitted and
- information being passed from one part of your application to another.

Listing 8: Examples of Events

In short, an event can be any action that occurs either internal or external to your processor. You associate an event with the rest of your application (primarily tasks, but also ISRs and background code) through the RTOS event services. The interaction between events and tasks follows certain simple rules:

- *Creating* an event makes it available to the rest of your system. You cannot signal an event, nor can any task(s) wait on the event, until it has been created. Events can be created with different initial values.
- Once an event has been created, it can be *signaled*. When an event is signaled, we say that the event has occurred. Events can be signaled from within a task or other background code, or from within an ISR. What happens next is dependent on whether there are one or more tasks waiting on the event.
- Once an event has been created, one or more tasks can *wait* it. A task can only wait one event at a time, but any number of tasks can all wait the same event. If one or more tasks are waiting an event and the event is signaled, the highest-priority task or the first task to wait the event will become eligible to run, depending on how the RTOS implements this feature. If multiple waiting tasks share the same priority, the RTOS will have a well-defined scheme[11] to control which task becomes eligible.

One reason for running tasks in direct response to events is to guarantee that at any time the system can respond as quickly as possi-

[11] Generally LIFO or FIFO, i.e. the most recent task or the first task, respectively, to wait the event will become eligible when the event is signaled.

possible to an event. That's because waiting tasks consume no[12] processing power – they'll remaining waiting indefinitely, until the event they're waiting on finally occurs. Furthermore, you can tailor when the system acts on the event (i.e. run the associated task) based on its relative importance, i.e. based on the priority of the task(s) associated with the event.

The key to understanding multitasking's utility is to know how to structure the tasks in your application. If you're used to superloop programming, this may be difficult at first. That's because a common mode of thinking goes along the lines of "First I need to do this, then that, then the other thing, etc. And I must do it over and over again, checking to see if or when certain events have happened." In other words, the superloop system monitors events in a sequential manner and acts accordingly.

For event-driven multitasking programming, you may want to think along these lines: "What events are happening in my system, both internally and externally, and what actions do I take to deal with each event?" The difference here is that the system is purely *event-driven*. Events can occur repetitively or unpredictably. Tasks run in response to events, and a task's access to the processor is a function of its priority.[13] A task can react to an event as soon as there are no higher-priority tasks running.

Note Priorities are associated with tasks, not events.

In order to use events in your multitasking application, you must first ask yourself:

- what does my system do?
- how do I divide up its actions into separate tasks?
- what does each task do?
- when is each task done?
- what are the events?
- which event(s) cause each task to run?

Note Events need not be associated with tasks one-to-one. Tasks can interact with multiple events, and vice versa. Also, tasks that do not interact with any events are easily incorporated – but they are usually assigned low priorities, so that they only run when the system has nothing else to do.

12 Unless they're waiting with a timeout, which requires the timer.

13 Task priorities are easily incorporated into event-based multitasking.

Events and Intertask Communications

An RTOS will support a variety of ways to communicate with tasks. In event-based multitasking, for a task to react to an event, the event must trigger some sort of communication with the task. Tasks may also wish to communicate with each other. *Semaphores*, *messages* and *message queues* are used for intertask communication and are explained below.

Common to all these intertask communications are two actions: that of *signaling* (also called *posting* or *sending*) and *waiting* (also called *pending* or *receiving*). Each communication also requires an initialization (*creating*).

Note All operations involving semaphores, messages and message queues are handled through calls to the operating system.

Semaphores

There are two types of semaphores: *binary* semaphores and *counting* semaphores. A binary semaphore can take on only two values, 0 or 1. A counting semaphore can take on a range of values based on its size – for example, an 8-bit counting semaphore's value can range from 0 to 255. Counting semaphores can also be 16-bit or 32-bit. Figure 6 illustrates how we will represent semaphores and their values:

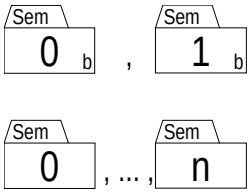


Figure 6: Binary and Counting Semaphores

Before it is used, a semaphore must be created with an initial value. The appropriate value will depend on how the semaphore is used.

Event Flags

Event flags are one such use for binary semaphores – they indicate the occurrence of an event. If a semaphore is initialized to 0, it means that the event has not yet occurred. When the event occurs, the semaphore is set to 1 by *signaling the semaphore*.

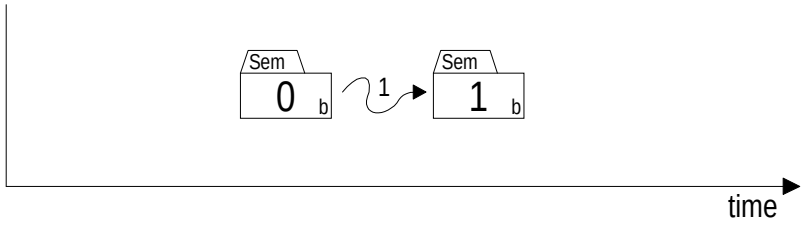


Figure 7: Signaling a Binary Semaphore

Figure 7 shows an ISR, task or other background code signaling [1] a binary semaphore. Once a semaphore (binary or counting) has reached its maximum value, further signaling is in error.

In addition to signaling a semaphore, a task can also *wait the semaphore*. Only tasks can wait semaphores – ISRs and other background code cannot. Figure 8 illustrates the case of an event having already occurred when the task waits the semaphore.

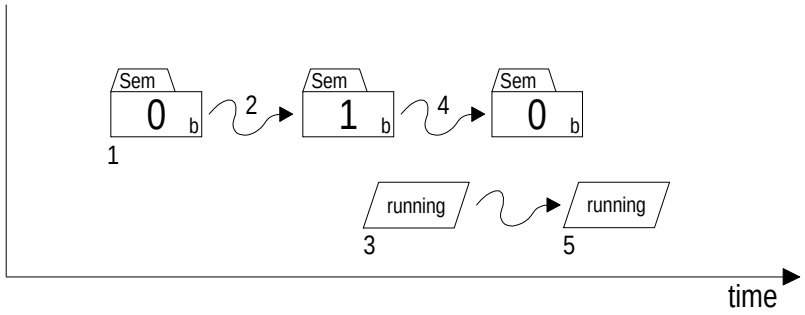


Figure 8: Waiting a Binary Semaphore When the Event Has Already Occurred

In Figure 8, the binary semaphore is initialized to 0 [1]. Some time later, the event occurs, signaling the semaphore [2]. When the task finally runs [3] and waits the semaphore, the semaphore will be reset [4] so that it can be signaled again and the task will continue running [5].

Note A semaphores is always initialized without any waiting tasks.

If the event has not yet occurred when the task waits the semaphore, then the task will be *blocked*. It will remain so (i.e. in the waiting state) until the event occurs. This is shown in Figure 9.

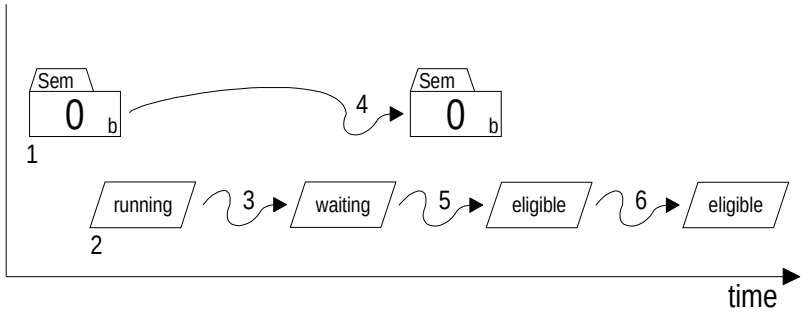


Figure 9: Signaling a Binary Semaphore When a Task is Waiting for the Corresponding Event

In Figure 9, an event has not yet been signaled [1] when a running task [2] waits the binary semaphore. Since the semaphore is not set, the task is blocked and must wait [3] indefinitely. The operating system knows that this task is blocked because it is waiting for a particular event. When the semaphore is eventually signaled from outside the task [4], the operating system makes the task eligible again [5] and it will run when it becomes the most eligible task [6]. The semaphore remains cleared because a task was waiting for it when it was signaled. Contrast this to Figure 7, where a semaphore is signaled with no tasks waiting for it.

It is also possible to combine event flags using the conjunctive (logical AND) or disjunctive (logical OR) combination of the event flags. The event is signaled when all (AND) or at least one (OR) of the event flags are set.

Note One or more tasks can concurrently wait an event. Which task becomes eligible depends on the operating system. For example, some operating systems may make the first task to wait the event eligible (FIFO), and others may make the highest-priority task eligible. Some operating systems are configurable to choose one scheme over the other.

Task Synchronization

Since tasks can be made to wait on an event before continuing, binary semaphores can be used as a means of *synchronizing* program execution. Multitask synchronization is also possible – Figure 10 shows two tasks synchronizing their execution via two separate binary semaphores.

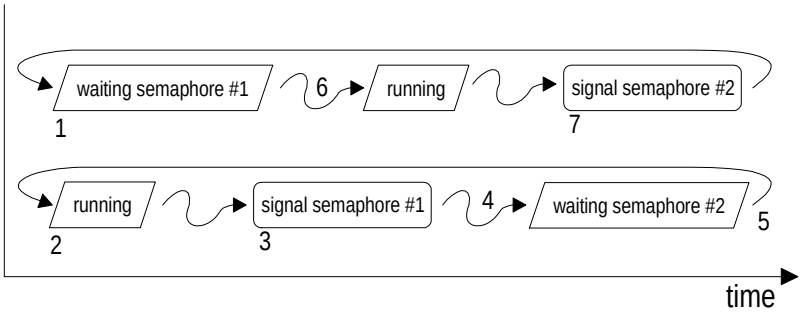


Figure 10: Synchronizing Two Tasks with Event Flags

In Figure 10, binary semaphores #1 and #2 are initialized to 0 and 1, respectively. The upper task begins by waiting semaphore #1, and is blocked [1]. The lower task begins running [2], and when it is ready to wait for the upper task it signals semaphore #1 [3] and then waits semaphore #2 [4], and is blocked [5] since it was initialized to 0. The upper task then begins running [6] since semaphore #1 was signaled, and when it is ready to wait for the lower task it signals semaphore #2 [7] and then waits semaphore #1, and is blocked [1]. This continues indefinitely. Listing 9 shows the pseudocode for this example.

```
initialize binary semaphore #1 to 0;
initialize binary semaphore #2 to 1;

UpperTask()
{
  for (;;) {
    /* wait for LowerTask() */
    wait binary semaphore #1;
    do stuff;
    signal binary semaphore #2;
  }
}

LowerTask()
{
  for (;;) {
    do stuff;
    signal binary semaphore #1;
    /* wait for UpperTask() */
    wait binary semaphore #2;
  }
}
```

Listing 9: Task Synchronization with Binary Semaphores

Resources

Semaphores can also be used to manage resources via *mutual exclusion*. The resource is available if the binary semaphore is 1, and is not available if it is 0. A task that wishes to use the resources must *acquire* it by successfully waiting the binary semaphore. Once it has acquired the resource, the binary semaphore is 0, and therefore any other tasks wishing to use the resource must wait until it is *released* (by signaling the binary semaphore) by the task that has acquired the resource.

```
initialize binary semaphore to 1;

TaskUpdateTimeDate()
{
  for (;;) {
    …
    prepare time & date string;
    wait binary semaphore;
    write time & date string to display;
    signal binary semaphore;
    …
  }
}

TaskShowAlert()
{
  for (;;) {
    wait binary semaphore;
    write alert string to display;
    signal binary semaphore;
  }
}
```

Listing 10: Using a Binary Semaphore to Control Access to a Resource

In Listing 10 a binary semaphore is used to control access to a shared resource, a display (e.g. an LCD). In order to enable access to it, the semaphore must be initialized to 1. A task wishing to write to the display must acquire the resource by waiting the semaphore. If the resource is not available, the task will be blocked until the resource is released. After acquiring the resource and writing to the display, the task must then release the semaphore by signaling it.

Resources can also be controlled with counting semaphores. In this case, the value of the counting semaphore represents *how many of the resources* are available for use. A common example is that of a ring buffer. A ring buffer has space for `m` elements, and elements are added to and removed from it by different parts of an application. Figure 11 shows a scheme to transmit character strings via

RS-232 using a counting semaphore to control access to a ring buffer.

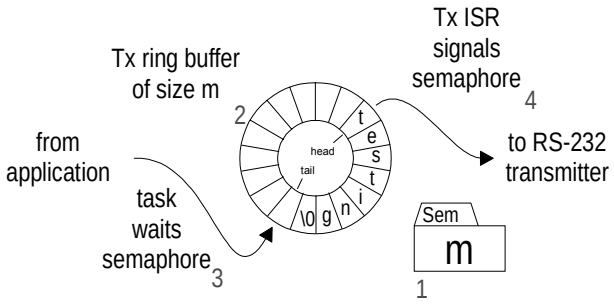


Figure 11: Using a Counting Semaphore to Implement a Ring Buffer

In Figure 11 a counting semaphore is initialized to `m` [1] to represent the number of spaces available in the empty ring buffer [2]. The ring buffer is filled at its tail[14] by the task [3] and emptied from its head by the ISR [4]. Before adding a character to the buffer the task must wait the semaphore. If it is blocked, it means that the buffer is full and cannot accept any more characters. If the buffer is not full, the semaphore is decremented, the task places the character at the tail of the buffer and increments the tail pointer. Once there are characters in the buffer[15], for each character the Tx ISR will remove it from the buffer, transmit it and increment the semaphore by signaling it. The corresponding pseudocode is shown[16] in Listing 11.

```
initialize counting semaphore to m;

TaskFillTxBuffer()
{
  for (;;) {
    wait semaphore;
    place char at TxBuff[tail pointer];
    increment tail pointer;
  }
}

ISRTxChar()
{
  send char at TxBuff[head pointer] out RS-232;
  increment head pointer;
  signal semaphore;
```

14 The tail pointer points to the next available free space for insertion into the ring buffer. The head pointer points to the first available element for removal from the ring buffer.

15 This is usually signified by enabling transmit interrupts.

16 The control of Tx interrupts, which varies based on transmitter configurations, is not shown.

```
}
```

Listing 11: Using a Counting Semaphore to Control Access to a Resource

By using a task to fill the ring buffer, the application need not poll the buffer's status at regular intervals to determine when to insert new characters. Nor does the application need to wait in a loop for room to insert characters into the buffer. If only part of a string is inserted before the task is blocked (i.e. the string is larger than the available room in the buffer), the task will automatically resume inserting additional characters each time the ISR signals the counting semaphore. If the application sends strings infrequently, a low task priority will probably suffice. Otherwise a high task priority may be necessary.

Note The RAM required for the semaphore that is used to manage a resource is separate from the RAM allocated to the resource itself. The RTOS allocates memory for the semaphore – *the user must allocate memory for the resource*. In this example, 8-bit counting semaphores limit the size of the ring buffer to 256 characters. The semaphore will require one byte of RAM irrespective of the actual (user-declared) size of the ring buffer itself.

Messages

Messages provide a means of sending arbitrary information to a task. The information might be a number, a string, an array, a function, a pointer or anything else. Every message in a system can be different, as long as both the sender and the recipient of the particular message understand its contents. Even the type of message can even change from one message to the next, as long as the sender and recipient are aware of this! As with semaphores, the operating system provides the means to create, signal and wait messages.

In order to provide general-purpose message contents, when a message is sent and received, the actual content of the message is not the information itself, but rather a *pointer to the information*. A pointer is another word for the *address* (or location) *of the information*, i.e. it tells where to find the information. The message's recipient then uses the pointer to obtain the information contained in the message. This is called *dereferencing* the pointer.[17]

[17] In C, `&` is the address of operator, and `*` is the unary operator for indirection. Therefore if `var` is a variable and `p` points to it, then `p=&var` and `*p` is equal to `var`.

If a message is initialized to be empty, it contains a *null pointer*. A null pointer is a pointer with a value of 0. By convention, a null pointer doesn't point to anything; therefore it carries no other information with it. A null pointer cannot be dereferenced.

Signaling (i.e. sending) a message is more complex than signaling a semaphore. That's because the operating system's message-signaling function requires a message pointer as an argument. The pointer passed to the function must correctly point to the information you wish to send in the message. This pointer is normally non-zero, and is illustrated in Figure 12.

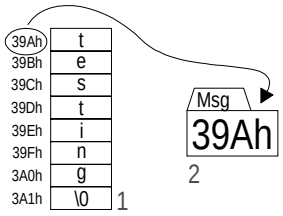


Figure 12: Signaling a Message with a Pointer to the Message's Contents

In Figure 12, a C-language character string[18] [1] is sent in a message [2] by signaling the message with a pointer. The string resides at a particular physical address. The message does *not* contain the first character of the string – it contains the address of the first character of the string (i.e. a pointer to the string), and the pointer's value is 39Ah. The pseudocode for sending this message is shown in Listing 12.

```
string[] = "testing";
p = address(string);
signal message with p;
```

Listing 12: Signaling a Message with a Pointer

To receive a message's contents, a task must wait the message. The task will be blocked until the message arrives. The task then extracts the contents of the message (i.e. the pointer) and uses the pointer in whatever manner it chooses. In Listing 13, the receiving task capitalizes the string that the message points to.

```
TaskCaps()
{
  for (;;) {
    wait message containing string pointer p;
```

[18] In C, character strings end with the NUL character ('\0').

```
    while ((p) is not null) { [19]
      if ('a' <= (p) <= 'z')
      (p) = (p) - 32;
      increment p;
    }
  }
}
```

Listing 13: Receiving a Message and Operating on its Contents

A message can contain at most one item of information (i.e. a pointer) at a time. If the message is empty, it can be signaled. If it's full, the message cannot be sent.

Messages can be used like binary semaphores. A message containing a null pointer is equivalent to a binary semaphore of value 0, and a message containing a non-zero pointer is equivalent to a binary semaphore of value 1. This is useful if binary semaphores are not explicitly supported by the RTOS.

Message Queues

Message queues are an extension of messages. A message queue can contain multiple messages (up to a predetermined number) at any time. Sending messages can continue until the message mailbox is full. A task that waits the message queue will receive messages until the message queue is empty.

An RTOS will need to allocate some additional RAM to manage each message queue. This RAM will be used to keep track of the number of messages in the message queue, and the order in which the messages exist in the message queue.

Summary of Task and Event Interaction

Here is a summary of the rules governing the interaction of tasks and events (i.e. semaphores, messages and message queues).

- An events must be initialized. It is initialized without any waiting tasks.
- A task cannot wait an event until the event has been initialized.
- Multiple tasks can wait a single event.
- A task can only wait one event at a time.

---

[19] "(pointer)" is pseudocode for "what is pointed to by the pointer."

- A semaphore's value can range from 0 to its maximum value, depending on its size.
- A message contains a pointer to some information.
- Message queues can hold multiple messages at once.
- An ISR, a task or other background code can signal an event.
- Only a task can wait an event.
- A task will be blocked (i.e. it will change to the waiting state) if the event it waits is not available.
- Which waiting task becomes eligible when an event is signaled is dependent on how the operating system implements event services.
- If an event has already been signaled, no task is waiting it, and it is signaled again, then either an error has occurred or the signaling task can be blocked. This is dependent on how the operating system implements event services.

Conflicts

A variety of conflicts may occur within a multitasking environment. They are described below.

Deadlock

Deadlock occurs with two or more tasks when each task is waiting for a resource controlled by another task. Since all of the affected tasks are waiting, there is no opportunity for any of the resources to become available. Therefore all the tasks will be deadlocked, i.e. they will wait indefinitely.

The solution is for all tasks wishing to acquire the resources to

- always acquire the resources in a predetermined order,
- acquire all the resources before continuing, and
- release the resources in the opposite order.

By using a *timeout* one can break a deadlock. When attempting to acquire the resource, an optional time period can be specified. If the resource is not acquired within that time period, the task continues, but with an error code that indicates that it timed out waiting for the resource. Special error handling may then be invoked.

Priority Inversions

Priority inversions occur when a high-priority task is waiting for a resource controlled by a low-priority task. The high-priority task must wait until the low-priority task releases the resource, whereupon it can continue. As a result, the priority of the high-priority task is effectively reduced to that of the low-priority task.

There are a variety of ways to avoid this problem (e.g. *priority inheritance*), most of which involve dynamically changing the priority of a task that controls a resource based on the priority of tasks wishing to acquire the resource.

RTOS Performance

The code to implement a multitasking RTOS may be larger than what's required in a superloop implementation. That's because each task requires a few extra instructions to be compatible with the scheduler. Even so, a multitasking application is likely to have much better performance and be more responsive than one with a superloop. That's because a well-written RTOS can take advantage of the fact that tasks that are not running often need not consume any processing power at all. This means that instead of spending instruction cycles testing flags, checking counters and polling for events, your multitasking application makes the most of the processor's power by using it directly where you need it most – on the highest-priority task that's eligible to run.

A Real-World Example

Let's look at an interesting example application – the controller for a remote soda-can vending machine. It must indicate (via LEDs on the buttons) if any selections are empty, handle the insertion of coins and bills, properly interpret the customer's selection, release the right item to the customer, and make change properly. A modern, microprocessor-controlled vending machine might also regulate internal temperatures (e.g. for soda cans), be connected to a network to relay out-of-stock information to a remote location, and be tied into a security system to deter vandalism. And of course all of this has to be done without error regardless of how many unpredictable things the customer does in the quest to quench his or her hunger or thirst.

The Conventional Superloop Approach

The refrigerated, vandal-resistant vending machine in our example has a user interface consisting of an array of item-selection buttons and slots for bills and coins. The main loop for a pseudo-code version of a traditional superloop implementation might look like this:

```
Initialize();
do forever
{
  ControlTemps();
  ShowEmpties();

  AcceptCurrency();

  flagSelectionGood = FALSE;
  ReadButtons();

  if ( flagSelectionGood ) {
    ReleaseItem();
    MakeChange();
  }

  if ( Tilt() )
    CallPolice();
}
```

Listing 14: Vending Machine Superloop

where some ISRs (not shown) are employed to do things like debounce the button presses. Listing 14 also shows neither the individual functions (e.g. `ReleaseItem()`) nor the global variables required to pass information between the functions, e.g. between `ReadButtons()` and `ReleaseItem()`.

Let's examine Listing 14 in more detail. In the superloop we call `ControlTemps()` once each time through the loop. On an 8-bit, 8MHz processor likely to be used in such an application, we might expect `ControlTemps()` to be called once every 200 microseconds when there's no user activity. This is a huge waste of processing power, as we know that we really only need to call it once a minute. We're calling `ControlTemps()` 5,000 times more often than necessary! While this may be acceptable in a vending machine, it's unlikely to be in a more demanding application.

One approach to fixing this would be to dedicate a periodic interrupt to set a globally visible bit every second. Then we could check this bit and call `ControlTemps()` when the bit is set high. This approach isn't too clever, because we're still doing an operation (testing the bit) every 200 microseconds. Another approach would be to move `ControlTemps()` completely into an ISR that's called

every second, but that's ill-advised, especially if `ControlTemps()` is a large and complex function.

In our example, `ReleaseItem()` will run only when money's in the machine and a button has been pressed. In other words, it's waiting for an event – an event characterized by the presence of the proper amount of money AND a valid selection button being pressed.

As illustrated in Listing 14, foreground / background superloop software designs puts most of the required processing in a single main loop that the processor executes over and over again. External events and time-critical processing are handled in the foreground via ISRs. Note that no single operation in the superloop has priority over any other. The execution of the functions proceeds in a rigidly serial manner, with the use of many hierarchical loops. When adding more functionality to a system like this, the main loop is likely to grow larger and slower, perhaps more ISRs will be needed, and system complexity will increase in your attempt to keep everything working as a whole.

For instance, in the above example there's no way for the customer to cancel a purchase. How would you modify the code to handle this additional requirement? You could write an expanded state machine to handle various scenarios, or use lots of timer interrupts to control how often various functions can run. But do you think someone else would understand what you wrote? Or even you, two years from now?

The Event-Driven RTOS Approach

If we start to talk about understanding, modifying and maintaining foreground / background code of moderate to severe complexity, it loses its appeal. That's because there are no clear relationships among the various functions in the superloop, nor between the functions and the flag variables, nor between the ISRs and the super loop. Let's try a different, task- and event-based approach.

Here's a list of tasks we can identify from the example above:

- Monitor and control internal temperature – `ControlTemps()`
- Display empty bins via LEDs – `ShowEmpties()`
- Accept or reject currency, and total it – `AcceptCurrency()`
- Debounce and read buttons – `ReadButtons()`
- Make change – `MakeChange()`

- Release selected item to customer – `ReleaseItem()`
- Attempt to protect the vending machine from vandalism – `CallPolice()`

Let's examine each of these tasks in a little more detail. We'll look at how important each one is, from 1 (most important) to 10 (least important), and when each task should run.

`ControlTemps()` is obviously important, as we want to keep the sodas cool. But it probably doesn't have to run more often than, say, once a minute, to accurately monitor and be able to control the temperature. We'll give it a priority of 4.

`ShowEmpties()` isn't too important. Moreover, the status of the empty bins only changes each time an item is released to the customer. So we'll give it a priority of 8, and we'd like it to run initially and once for every time an item is released.

`ReadButtons()` should have a reasonably high priority so that there's no noticeable lag when the customer presses the machine's buttons. Since button presses are completely asynchronous, we want to test the array of buttons regularly for activity. Let's give it a priority of 3, and run it every 40 milliseconds.

Since `AcceptCurrency()` is also part of the user interface, we'll give it the same priority as `ReadButtons()` and we'll run it every 20 milliseconds.

The machine's manufacturer does not consider `MakeChange()` to be all that important, so we'll give it a priority of 10. We'll link it to `ReleaseItem()`, since change must be made only after the selected item is delivered to the customer.

`ReleaseItem()` is interesting because we only need it once the proper amount of money has been accepted and an item button is pressed. To respond quickly we'll give it a priority of 2, and we'd like it to run when the above combination of money and button press occurs.

The machine's manufacturer makes a big point of how vandal-resistant it is. It's even capable of detecting an attack (through built-in tilt sensors) and calling the local security service. We'll give `CallPolice()` the highest priority of 1, and we'll check the tilt sensors every 2 seconds for an attack.

Step By Step

Our vending machine example requires seven tasks with six different priorities, and a timer resolution of 20ms. To create this multitasking application from these functions, we'll need to:

- initialize the operating system,
- modify the structure of the tasks so as to be compatible with the operating system and the events,
- create prioritized tasks from the task functions,
- link the real-world events to events that the operating system understands,
- create a system timer to keep track of elapsed time,
- start the various tasks and
- begin multitasking.

Initializing the Operating System

Initializing the operating system is usually straightforward, e.g.

```
InitializeMultitasking();
```

This creates the necessary (empty) structures the operating system will use to manage task execution and events. At this point all of the system's tasks are in the *uninitialized / destroyed* state.

Structuring the Tasks

The tasks written for a multitasking application look similar to those written for a superloop application. The big difference lies in the overall program structure. The multitasking tasks are not contained in any loops or larger functions – they're all independent functions. `ReleaseItem()`, which releases an item once a set of conditions has been met, might look like this in pseudocode:

```
ReleaseItem()
{
  do forever {
    WaitForMessage(messageSelection, item);

    Release(item);
  }
}
```

Listing 15: Task Version of ReleaseItem()

In Listing 15 `ReleaseItem()` waits forever for a (particular) message and does nothing until the message arrives. While it's waiting

for the message to arrive, `ReleaseItem()` is in the *waiting* state. When the message is sent, `ReleaseItem()` becomes eligible to run, and when it runs, it extracts the contents of the message (in this case, a code for the desired item, e.g. "B3") and releases it to the customer. `ReleaseItem()` is not inside any larger loop, nor is it called by any other functions (except indirectly by the scheduler, below).

`CallPolice()` has a similar "stand-alone" look to it:

```
CallPolice()
{
  do forever {
    Delay(1000);

    if ( Tilt() )
      SendMsgToPoliceHQ();
  }
}
```

Listing 16: Task Version of CallPolice()

`CallPolice()` enters an infinite loop where it delays itself for 1000 x 20ms, or 2 seconds, and then sends a message to the police headquarters if the vending machine's tilt sensors detect an attack. It repeats this sequence indefinitely. While delayed, `CallPolice()` is in the *delayed* state.

Prioritizing the Tasks

An operating system call assigns a priority to a task, and prepares the task for multitasking. For example,

```
CreateTask(ShowEmpties(), 8)
```

Listing 17: Prioritizing a Task

tells the operating system that it should give `ShowEmpties()`a priority of 8 and add it to the tasks whose execution it will manage. `ShowEmpties()` is now in the *stopped* state.

Interfacing with Events

In Listing 15, `ReleaseItem()` is using a message to handle an event – namely the release of an item. That message needs to be initialized:

```
CreateEvent(messageSelection, empty);
```

Listing 18: Creating a Message Event

By initializing `messageSelection` to `empty` (i.e. no valid selection has been made), `ReleaseItem()` will only release an item once the required events (enough money inserted and appropriate button pressed) have occurred.

Adding the System Timer

An RTOS needs some way to keep track of real time – this is usually provided via some sort of timer function that the application must call at a regular, predefined rate. In this case that rate is 50Hz or every 20ms. Calling the system timer is often accomplished through an interrupt, e.g.:

```
InterruptEvery20ms()
{
  SystemTimer();
}
```

Listing 19: Calling the System Timer

Starting the Tasks

Applications must create all of their tasks and events before any of them are actually used. By providing an explicit means of starting tasks, the RTOS enables you to manage system startup in a predictable way:

```
StartTask(ControlTemps());
StartTask(ShowEmpties());
StartTask(AcceptCurrency());
StartTask(ReadButtons());
StartTask(MakeChange());
StartTask(ReleaseItem());
StartTask(CallPolice());
```

Listing 20: Starting all Tasks

Since multitasking has not yet started, the order in which tasks are started is immaterial and is not in any way dependent on their priorities. At this point all of the tasks are in the *eligible* state.

Enabling Multitasking

Once everything is in place, events have been initialized and the tasks have been started (i.e. they are all ready to execute), multitasking can begin:

```
StartMultitasking();
```

Listing 21: Multitasking Begins

The scheduler will take the eligible task with the highest priority and run it – i.e. that task will be in the *running* state. From now on, the scheduler will ensure that the highest-priority task is the only one running at any time.

Putting It All Together

Listing 22 is a complete listing of the task- and event-driven vending machine application in pseudocode:

```
#include "operatingsystem.h"

extern AlertPoliceHQ()
extern ButtonPressed()
extern DisplayItemCounts()
extern InterpretSelection()
extern NewCoinsOrBills()
extern PriceOf()
extern ReadDesiredTemp()
extern Refund()
extern ReleaseToCustomer()
extern SetActualTemp()
extern Tilt()

ControlTemps()
{
  do forever {
    Delay(500);

    ReadActualTemp();
    SetDesiredTemp();
  }
}


ShowEmpties()
{
  DisplayItemCounts();

  do forever {
    WaitForSemaphore(semaphoreItemReleased);

    DisplayItemCounts();
  }
```

```
}


AcceptCurrency()
{
  do forever {
    Delay(1);

    money += NewCoinsOrBills();
  }
}


ReadButtons()
{
  do forever {
    Delay(2);

    button = ButtonPressed();
    if ( button ) {
      item = InterpretSelection(button);
      SignalMessage(messageSelection, item);
    }
  }
}


MakeChange()
{
  do forever {
    WaitForMessage(messageCentsLeftOver, change);

    Refund(change);
  }
}


ReleaseItem()
{
  CreateEvent(semaphoreItemReleased, 0);
  CreateEvent(messageCentsLeftOver, empty);

  do forever {
    WaitForMessage(messageSelection, item);

    if ( money >= PriceOf(item) ) {
      ReleaseToCustomer(item);
      SignalSemaphore(semaphoreItemReleased);
      SignalMessage(messageCentsLeftOver,
        money - PriceOf(item));
        money = 0;
    }
  }
}


CallPolice()
```

```
{
  do forever {
    Delay(1000);

    if ( Tilt() )
      AlertPoliceHQ();
  }
}


InterruptEvery20ms()
{
  SystemTimer();
}


main()
{
  money = 0;

  InitializeMultitasking();

  CreateTask(ControlTemps(),   4)
  CreateTask(ShowEmpties(),    8)
  CreateTask(AcceptCurrency(), 3)
  CreateTask(ReadButtons(),    3)
  CreateTask(MakeChange(),    10)
  CreateTask(ReleaseItem(),    2)
  CreateTask(CallPolice(),     1)

  CreateEvent(messageSelection, empty);

  StartTask(ControlTemps());
  StartTask(ShowEmpties());
  StartTask(AcceptCurrency());
  StartTask(ReadButtons());
  StartTask(MakeChange());
  StartTask(ReleaseItem());
  StartTask(CallPolice());

  StartMultitasking();
}
```

Listing 22: RTOS-based Vending Machine

The RTOS Difference

The program in Listing 22 has an entirely different structure than the superloop one in Listing 14. Several differences are immediately apparent:

- It's somewhat longer – this is mainly due to the overhead of making calls to the operating system.

- There are clearly-defined runtime priorities associated with each task.
- The tasks themselves have simple structures and are easy to understand. Those that communicate with other tasks or ISRs use obvious mechanisms (e.g. semaphores and messages) to do so. Initialization can be task-specific.
- The use of global variables is minimized.
- There are no delay loops.
- It's very easy to modify, add or delete a task without affecting the others.
- The overall behavior of the application is largely dependent on the task priorities and intertask communication.

Perhaps most importantly, the RTOS handles the complexity of the application automatically – tasks run on a priority basis, task switching and state changes are handled automatically, delays require a minimum of processor resources, and the mechanisms of intertask communications are hidden from view.

There are other differences that become more apparent during runtime. If we were to look at a timeline showing task activity, we would see

- Every 2 seconds `CallPolice()` wakes up to check for tampering and then returns to the delayed state,
- Every second `ControlTemps()` wakes up to adjust the internal temperature and then returns to the delayed state,
- Every 40ms `ReadButtons()` wakes up to debounce any button presses and then returns to the delayed state,
- Every 20ms `AcceptCurrency()` wakes up to monitor the insertion of coins and bills and then returns to the delayed state, and
- `ShowEmpties()`, `MakeChange()` and `ReleaseItem()` do nothing until a valid selection has been made, whereupon they briefly "come to life," deliver the selected item, refund any change and show full/empty item statuses, respectively, before returning to the waiting state.

In other words, for the vast majority of the time it's running, the vending machine's microcontroller has very little to do because the scheduler sees only delayed and waiting tasks. If the vending ma-

chine's manufacturer wanted to promote "Internet connectivity for enhanced stock management, remote querying and higher profits" as an additional feature, adding an extra task to transmit sales data (e.g. which sodas are purchased at what time and date and at what outside temperature) and run a simple web server would be as easy as creating another task to run in addition to the ones above and assigning it an appropriate priority.

Chapter 3 • Installation

Introduction

You will need access to the Salvo libraries and source files in order to build your Salvo application. Salvo also includes a large number of accompanying files of various types (e.g. test and demonstration programs, a User's Manual, etc.) – see *Appendix C • File and Program Descriptions* for more information.

You may receive Salvo in one of two forms: either on a CD-ROM, or in a self-extracting executable. Contained in each is a setup program (installer) for use on Wintel[20] development platforms, as well as the User's Manual and other files. Nearly all of the Salvo files are contained in compressed and encrypted form within the installer. You will need a valid serial number in order to extract them with the installer.

CD-ROM Installation

Note This section assumes you are installing Salvo onto a PC or PC compatible running Microsoft Windows 98. The installation for Windows 95, Windows 2000 and Windows NT is similar. If you are installing onto a PC running Windows 3.1, the installation is substantially similar, with some exceptions as regards path and filenames.

1. Run Windows and insert the Salvo CD-ROM into your PC's CD-ROM drive.

2. If you have auto insert notification[21] enabled, you should see the Salvo splash screen automatically after a few seconds. If not, open the CD-ROM by double-clicking on the CD-ROM drive's icon. Alternatively, you can launch the installer by selecting

Start > Run...

[20] A Wintel machine is an x86-type PC running Microsoft Windows 3.1/95/98/2000/NT.

[21] See My Computer > Properties > Device Manager > CD-ROM > Properties > Settings.

and typing

```
drive:\SETUP.EXE
```

where `drive` the is the drive letter for your CD-ROM drive (e.g. `D`) and then clicking on the **OK** button. The Welcome screen appears:

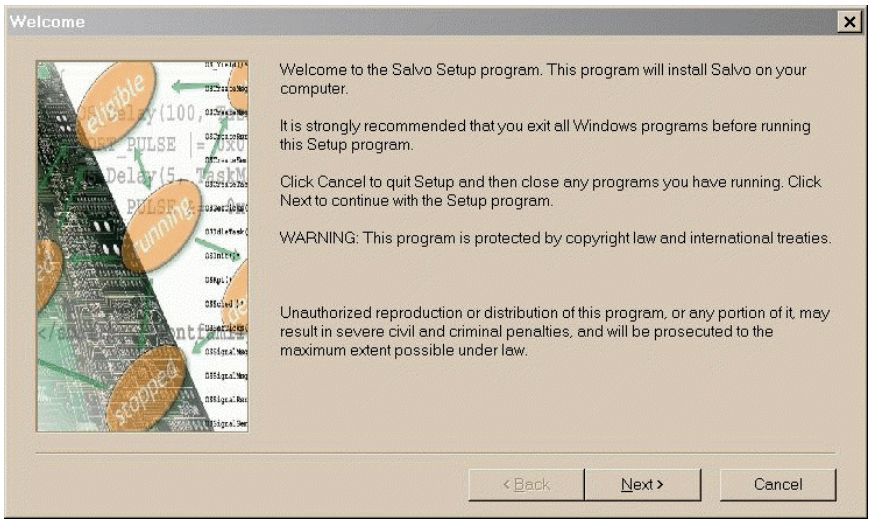


Figure 13: Welcome Screen

Note Most of the installer's screens contain **Next**, **Back** and **Cancel** buttons. Click on the **<Back** button for the previous screen. Click on the **Cancel** button to abort the installation.

3. Click on the **Next>** button. The Important Notes screen appears:

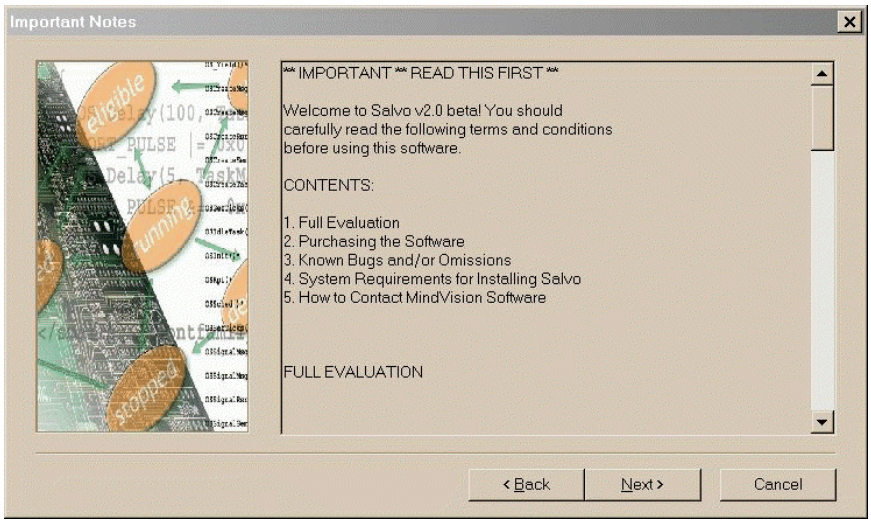


Figure 14: Important Notes Screen

This screen contains the `README.TXT` document. Read this document before you continue installing Salvo. This document is included in the Salvo folder once the installation is complete.

4. Click on the **Next>** button. The Salvo License Agreement screen appears:

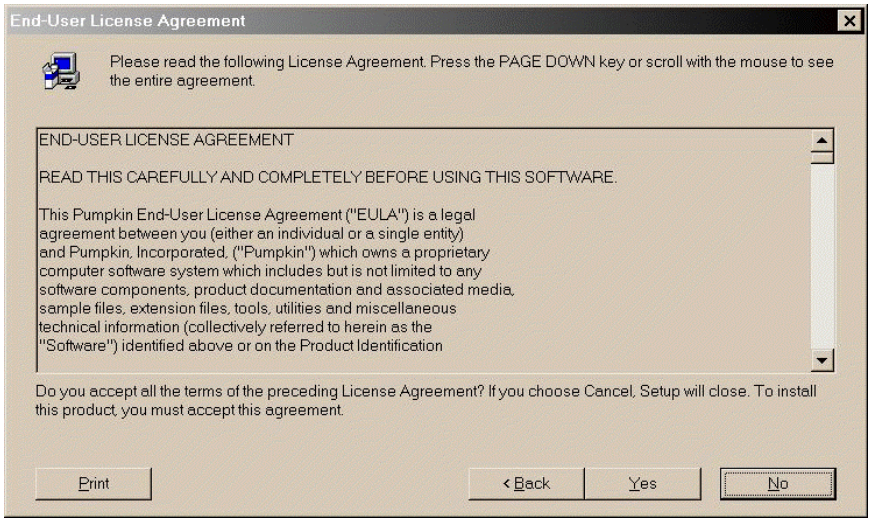


Figure 15: Salvo License Agreement Screen

This screen contains the Pumpkin Salvo License Agreement. Read this agreement carefully. This document is included in the Salvo folder once the installation is complete. You must accept the terms of the License in order to continue installing Salvo. You can print a copy of the License by clicking on the **Print** button. To accept the License, click on the **Yes** button. If you do not accept the License, click on the **No** button and return the software.[22]

5. After you click on the **Yes** button, the Registration screen appears:

22 Instructions on returning the software are contained in the License and in the User's Manual.

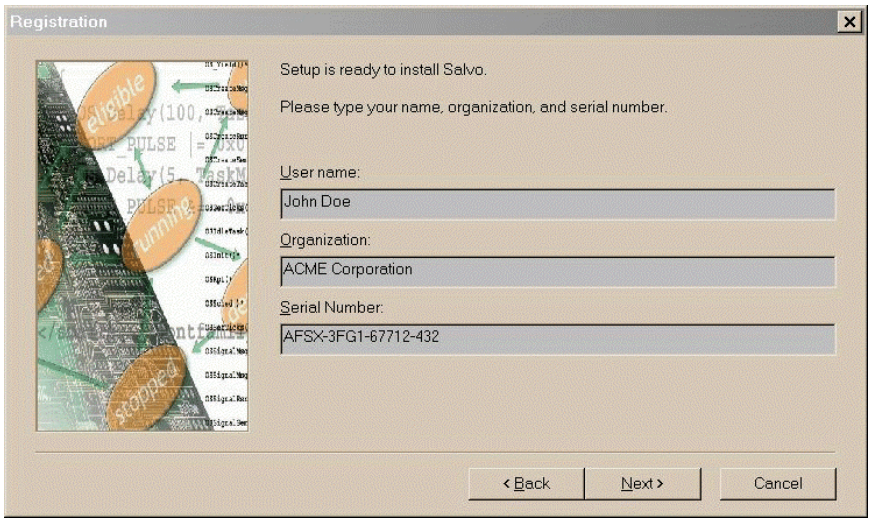


Figure 16: Registration Screen

Enter your user name, the organization you belong to (if applicable), and the serial number. The serial number can be found inside the Salvo packaging.

Note The letters in the serial number are case-sensitive.

6. Once you've entered a valid serial number and clicked on the **Next>** button, the Choose Destination Location screen appears:

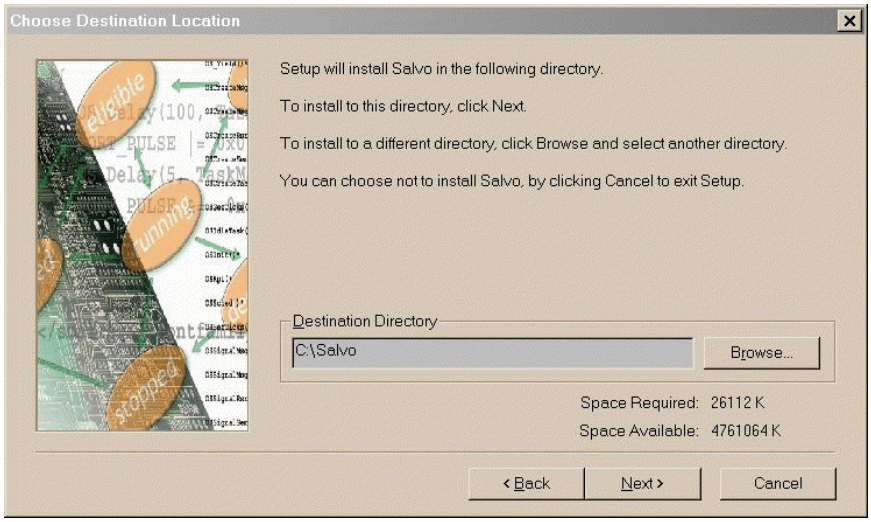


Figure 17: Choose Destination Location Screen

This screen allows you to set the directory where Salvo will be installed. The installer will place several[23] directories, some with nested subdirectories, in the destination directory. You can leave

[23] See Figure 21: Typical Salvo Destination Directory Contents.

the destination directory at its default (`C:\Salvo`) or you can change it by clicking on the **Browse…** button and selecting a different destination directory.

Note In order to avoid potential compiler problems with long pathnames, we recommend that you choose a destination directory that is as close to the root directory of the destination drive as possible. Choosing a deeply nested directory (e.g. `C:\MyProjects\Programming\Tools\RTOS\Salvo`) may cause problems with DOS-based and other tools due to exceedingly long pathnames for Salvo files.

---

7. After clicking on the **Next>** button the Setup Type screen appears:

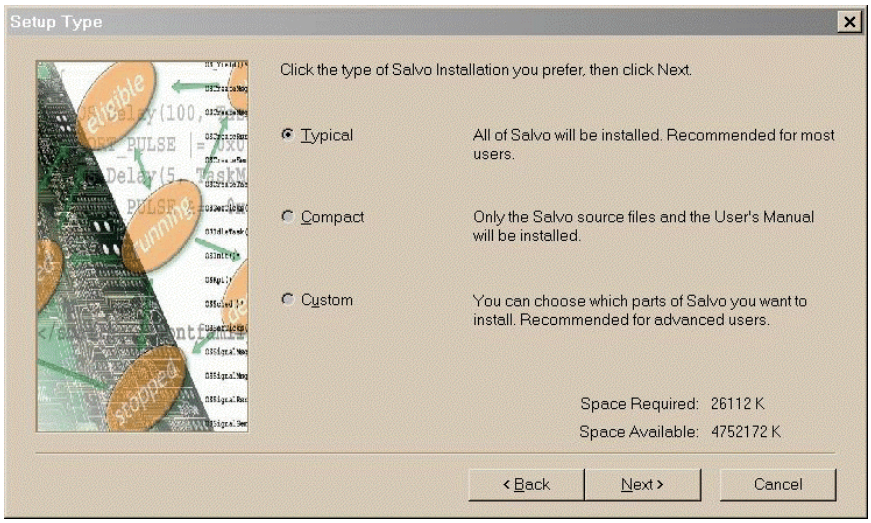


Figure 18: Setup Type Screen

You can choose from three different types of Salvo installations with this screen. Most users will choose the **Typical** setup, which installs all of Salvo. The **Compact** setup installs only the Salvo source files and the User's Manual. By choosing **Custom** you have complete control of what will be installed.

Tip If you ever accidentally modify and/or delete one or more Salvo source files, you can use the **Details** button in the Select Components screen of a Custom installation to specify the exact file(s) you want to restore / reinstall.

---

8. After choosing the type of installation, click on the **Next>** button and the Ready to Install screen appears:

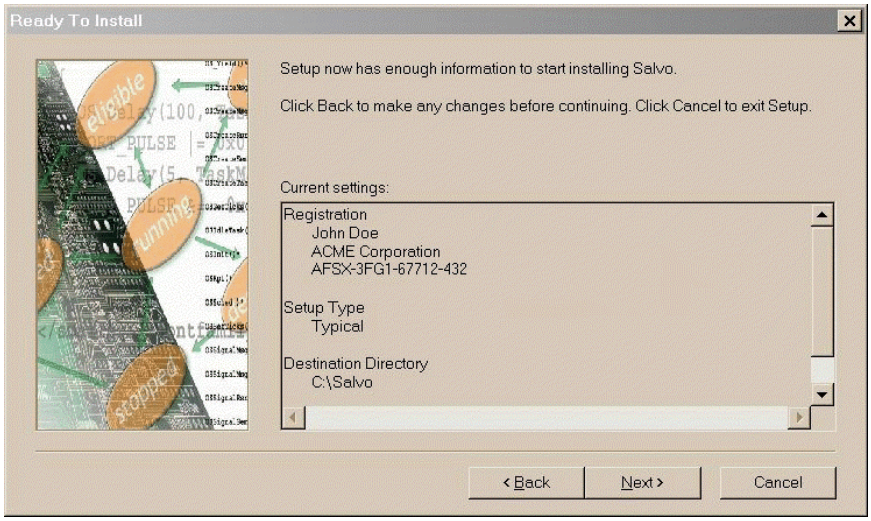


Figure 19: Ready To Install Screen

Verify that these settings are correct. If not, click on the **<Back** button and make the necessary changes. Click on the **Next>** button and the installer will place all of the Salvo files in their respective subdirectories of the destination directory. When it is done, the Finished screen appears:

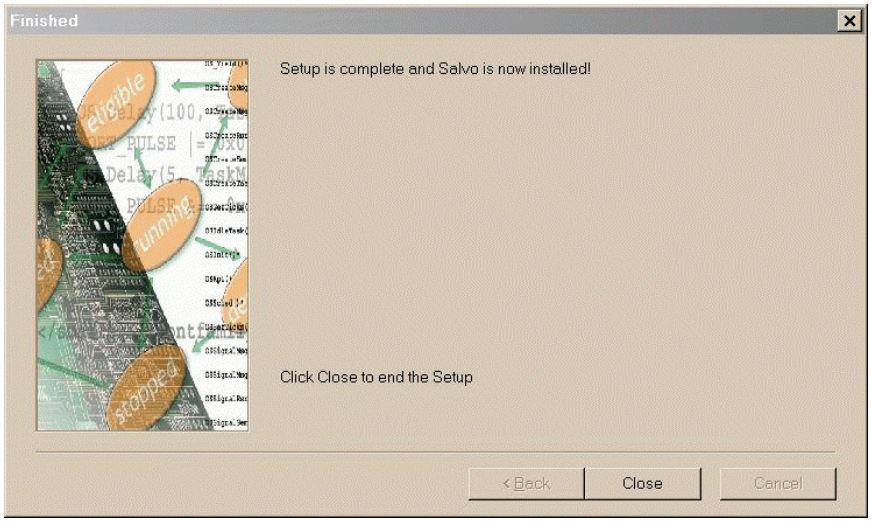


Figure 20: Finished Screen

9. Click on the **Close** button to finish the installation.

Internet Installation

If you have obtained Salvo via the Internet, launch the installer `SETUP.EXE` on your Wintel PC and follow steps 3 through 9 as outlined above.

Network Installation

If you are working in a networked environment with code sharing (e.g. for revision control) and need to install Salvo on a shared network disk, run the installer on a Wintel PC and choose a directory on a network drive as the destination directory. You may find it convenient to create the shortcuts in the Salvo Start Menu programs folder on each machine that is accessing Salvo over the network.

Note Network installations must comply with the terms of the Salvo License Agreement. See the License for more information.

Installing Salvo on non-Wintel Platforms

If you are developing Salvo applications on a non-Wintel platform, you will still need access to a Wintel machine in order to run the installer. The installer will place all of Salvo's files into the selected destination directory (the default is `C:\Salvo`), with multiple subdirectories. You can then copy the entire subdirectory to another machine via a network or a mass storage device (e.g. Zip, Jaz, tape, etc.).

Note The Salvo License Agreement allows only one copy of the Salvo directories per installation. You must remove the entire Salvo directory from the Wintel machine after you have transported it to your non-Wintel development environment. See the License for more information.

Alternatively, if you are working in a networked environment with cross-platform file sharing, you can run the installer on a Wintel PC and select a (remote) directory on your non-Wintel platform as the destination directory for the installation. All of the Salvo files will be installed to the remote directory. After the installation is complete you may want to remove the Start Menu items from the Wintel PC if you will not be using them.

Installing Salvo Lite

Installing the freeware version of Salvo is just like installing the full version. The Salvo Lite distribution contains a subset of the files found in the full-version distribution. A separate installer (salvo-lite-version-target.exe) is provided with this version, and it does not require a serial number.

A Completed Installation

Your Salvo directory should look similar to this after a typical installation:

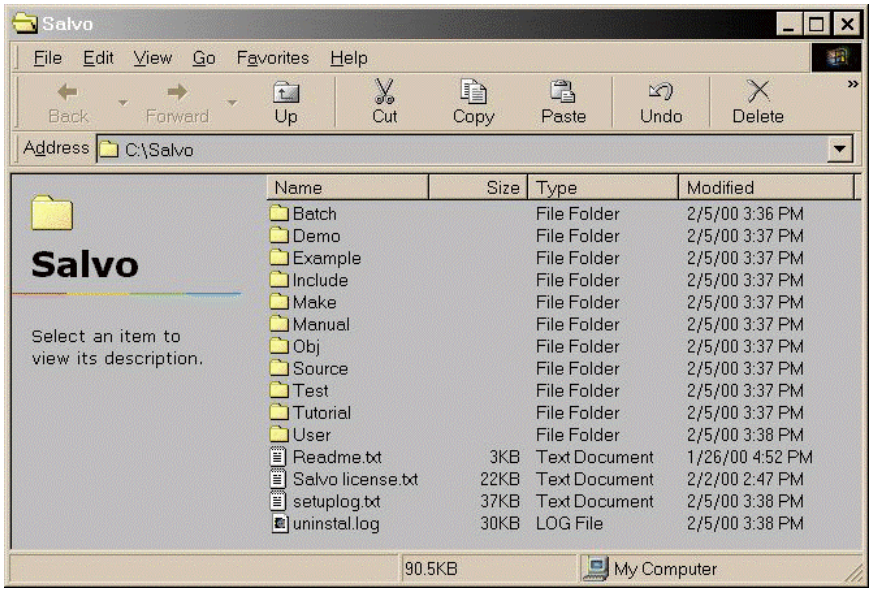


Figure 21: Typical Salvo Destination Directory Contents

The setup program also adds a Salvo folder to the Start Menu programs:



Figure 22: Start Menu Programs Folder

Shortcuts are provided to the Salvo directory and to the Salvo User's Manual. A URL to Salvo's web site is also provided. The Salvo Start Menu programs folder also contains a shortcut for removing Salvo from your PC – See Uninstalling Salvo, below.

Uninstalling Salvo

For Wintel machines, the setup program automatically provides an uninstaller. To use the uninstaller, select Uninstall Salvo as shown below:

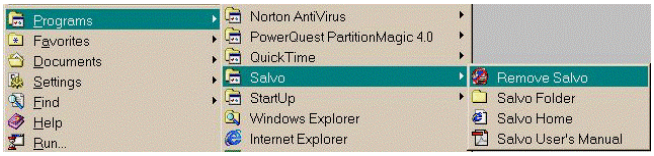


Figure 23: Launching the Uninstaller

When prompted by the uninstaller, click on the **Yes** button to confirm file deletion:

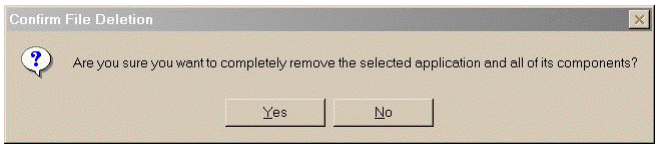


Figure 24: Confirm File Deletion Screen

The uninstaller will display the following screen upon successfully removing Salvo from your development platform:

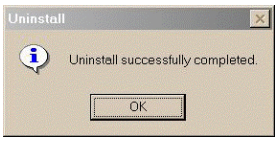


Figure 25: Uninstall Complete Screen

Click on the **OK** button to finish uninstalling Salvo.

Uninstalling Salvo on non-Wintel Machines

If you are using Salvo on another platform (e.g. Linux), simply delete the Salvo destination directory and all of its subdirectories.

Chapter 4 • Tutorial

Introduction

In this chapter we'll use a two-part, step-by-step tutorial to help you create a Salvo application from scratch. The first part is an introduction to using Salvo to write a multitasking program in C. In the second part we'll compile it to a working application.

Part 1: Writing a Salvo Application

Let's create a multitasking Salvo application step-by-step, introducing various concepts and Salvo features as we go. We'll start with a minimal application in C and build on it. We'll explain the purpose and use of each new Salvo feature, and describe in-depth what's happening in the application.

Tip Each one of the C listings below is provided as a complete application in the `salvo\tut` directory, with projects, source code and executables. You may find them useful to gain more insight into their operation. All measured data in the tutorial are from Salvo Test System A. See *Chapter 9 • Performance* for more information on test systems.

Tip There's also a simple multitasking Salvo application in `salvo\demo\d4`. You may also find it useful if you're new to Salvo.

Initializing Salvo and Starting to Multitask

Each working Salvo application is a combination of calls to Salvo user services and application-specific code. Let's start using Salvo by creating a multitasking application.

A minimal Salvo application is shown in Listing 23. This program is located in `salvo\tut\tu1\main.c`.

```
#include "main.h"
#include <salvo.h>
```

```
int main( void )
{
  Init();

  OSInit();

  for (;;)
    OSSched();
}
```

Listing 23: A Minimal Salvo Application

This elementary program calls two Salvo user services whose function prototypes are declared in `salvo.h`. `OSInit()` is called once, and `OSSched()` is called over and over again from within an infinite loop.

Note `OSSched()` is in the `for()` loop, despite the lack of curly braces.

Tip All user-callable Salvo functions are prefixed by "`OS`" or "`OS_`".

Note The `Init()` function in `main()` is provided for device initialization.[24] It and the header file `main.h` have nothing to do with the Salvo code per se, but are provided for completeness.

OSInit()

`OSInit()` initializes all of Salvo's data structures, pointers and counters, and must be called before any other calls to Salvo functions. Failing to call `OSInit()` first before any other Salvo routines may result in unpredictable behavior.

OSSched()

`OSSched()` is Salvo's multitasking scheduler. Only tasks which are in the eligible state can run, and each call to `OSSched()` results in the most eligible task running until the next context switch within that task. In order for multitasking to continue, `OSSched()` must be called repeatedly.

Tip In order to make best use of your processor's call ... return stack, you should call `OSSched()` directly from `main()`.

In Depth

Since there are no tasks eligible to run, the scheduler in Listing 23 has very little to do.

[24] E.g. oscillator select and digital I/O crossbar select on Cygnal C8051F005 single-chip microcontroller.

Creating, Starting and Switching tasks

Multitasking requires eligible tasks that the scheduler can run. A multitasking Salvo application with two tasks is shown in Listing 24. This program is located in `salvo\tut\tu2\main.c`.

```
#include "main.h"
#include <salvo.h>

_OSLabel(TaskA1)
_OSLabel(TaskB1)

void TaskA( void )
{
  for (;;)
    OS_Yield(TaskA1);
}

void TaskB( void )
{
  for (;;)
    OS_Yield(TaskB1);
}

int main( void )
{
  Init();

  OSInit();

  OSCreateTask(TaskA, OSTCBP(1), 10);
  OSCreateTask(TaskB, OSTCBP(2), 10);

  for (;;)
    OSSched();
}
```

Listing 24: A Multitasking Salvo Application with two Tasks

`TaskA()` and `TaskB()` do nothing but run and context switch over and over again. Since they both have the same priority (10), they run one after the other, continuously, separated by trips through the scheduler.

In order for multitasking to function properly, a running task must return control to the scheduler. This occurs via a context switch (or task switch) inside the task. Because it is designed to work without a stack, Salvo only supports context switching at the task level.

Warning A Salvo context switch at a call ... return level below that of the task (e.g. within a subroutine called by the task) will cause unpredictable behavior.

To multitask in Salvo, you must create and start tasks. Tasks are functions that consist of an optional initialization followed by an infinite loop containing at least one context switch. Salvo tasks cannot take any parameters. When the task is created via `OSCreateTask()`, you assign an unused *task control block* (*tcb*) to it and it is placed in the stopped state. A task can be created in many parts of your program. Tasks are often created prior to the start of multitasking, but they may also be created afterwards.

In order for a task to be able to run, it must be in the eligible state. `OSStartTask()` can make a stopped task eligible. However, in the interest of keeping the Salvo code size small, `OSCreateTask()` *automatically* starts the task that it has created.[25] Therefore a call to `OSStartTask()` is unnecessary. Once a task is made eligible, it will run by the scheduler as soon as it becomes the most eligible task, i.e. the eligible task with the highest priority.

Tip When a group of eligible tasks all share the same priority, they will execute one after the other in a round-robin fashion.

A stopped task can be started in many parts of your program. Tasks can only be started after they are created. A task may be started after multitasking begins.

OS_Yield()

Every task must context-switch at least once. `OS_Yield()` is Salvo's unconditional context switcher. A common place to find `OS_Yield()` would be at the bottom of, but still within, a task's infinite loop.

Note All Salvo user services with conditional or unconditional context switches are prefixed by "`OS_`".

Tip Each Salvo context switch requires a unique, explicit label. An easy way to create a label is to use Salvo's `_OSLabel()` macro, with a label name that you provide. Then, use that label as the label argument for the context switch. This is the purpose of the labels `TaskA1` and `TaskB1` above. `TaskA1` is the label of the first context switch within `TaskA()`. You may prefer an alternative naming convention, like `TaskA_label1`, and so on.

[25] Optionally, the task can be left in the stopped state by using `OSDONT_START_TASK`.

OSCreateTask()

To create a task, call `OSCreateTask()` with a *task starting address*, a *tcb pointer* and a *priority* as parameters. The starting address is usually the start of the task, specified by the task's name. Each task needs its own, unique tcb. The tcb contains all of the information Salvo needs to manage a task, like its start/resume address, state, priority, etc. There are `OSTASKS` tcbs available for use, numbered from `1` to `OSTASKS`. The `OSTCBP()` macro is a shorthanded[26] way of specifying a pointer to a particular Salvo tcb, e.g. `OSTCBP(2)` is a pointer to the second tcb. The task priority is between 0 (highest) and 15 (lowest), and need not be unique to the task. Once created, a task is in the stopped state.

The default behavior for `OSCreateTask()` is to also start the Salvo task with the specified tcb pointer by making it eligible. It may be a while before the task actually runs, depending on the priority of the task, the states of any higher-priority tasks, and when the scheduler will run again.

Tip Many Salvo services return error codes that you can use to detect problems in your application. See *Chapter 7 • Reference* for more information.

In Depth

Listing 24 illustrates some of the basic concepts of an RTOS – tasks, task scheduling, task priorities and context switching. Tasks are functions with a particular structure – infinite loops are commonly used. A task will run whenever it is the most eligible task, and the scheduler decides which task is eligible based on the task priorities. Since Salvo is a cooperative RTOS, each task must relinquish control back to the scheduler or else no other tasks will have a chance to run. In this example, this is accomplished via `OS_Yield()`. In the following examples, we'll use other context switchers in place of `OS_Yield()`.

While it's perhaps not immediately apparent, Listing 24 also illustrates another basic RTOS concept – that of the task state. In Salvo, all tasks start out as destroyed. Creating a task changes it to stopped, and starting a task makes it eligible. When the task is actually executing it's said to be running. In this example, after being created and started, each task alternates between eligible and running over and over again. And there's a short time period during iteration of the main `for()` loop where neither task is running, i.e. they're both eligible – that's when the scheduler is running.

---

[26] `&OStcbArea[n-1]` is the longhanded way.

Task scheduling in Salvo follows two very simple rules: First, whichever task has the highest priority will run the next time the scheduler is called. Second, all tasks with the same priority will run in a round-robin manner as long as they are the most eligible tasks. This means that they will run one after the other until they have all run, and then the cycle repeats itself.

Adding Functionality to Tasks

Listing 25 shows a multitasking application with two tasks that do more than just context switch. We'll use more descriptive task names this time. This program is located in `salvo\tut\tu3\main.c`.

```
#include "main.h"
#include <salvo.h>

_OSLabel(TaskCount1)
_OSLabel(TaskShow1)

unsigned int counter;

void TaskCount( void )
{
  for (;;) {
    counter++;

    OS_Yield(TaskCount1);
  }
}

void TaskShow( void )
{
  InitPORT();
  PORT = 0x00;

  for (;;) {
    PORT = (PORT & ~0xFE)|((counter >> 8) & 0xFE);

    OS_Yield(TaskShow1);
  }
}

int main( void )
{
  Init();

  OSInit();

  OSCreateTask(TaskCount, OSTCBP(1), 10);
  OSCreateTask(TaskShow,  OSTCBP(2), 10);

  counter = 0;
```

```
  for (;;)
    OSSched();
}
```

Listing 25: Multitasking with two Non-trivial Tasks

The two tasks in Listing 25 run independently of each other, and they both access a shared global variable, a 16-bit counter. The counter is initialized[27] before multitasking begins. The first task increments the counter every time it has a chance to run. The other task takes the counter and outputs the upper 7 bits to an 8-bit port (`PORT`) with 8 LEDs connected to it. This goes on indefinitely.

In Depth

In Listing 25, neither task actually runs until multitasking begins with the call to the Salvo scheduler. Each time `OSSched()` is called, it determines which task is most eligible to run, and transfers program execution to that particular task. Since both tasks have the same priority, and are equally eligible to run, it is up to Salvo to decide which task will run first.

In this particular example, `TaskCount()` will run first.[28] It will start by incrementing the counter, and will then context-switch via `OS_Yield()`. This macro will make a note of where program execution is in `TaskShow()` (it's at the end of the `for()` loop), and then return program execution to the scheduler. The scheduler then examines `TaskCount()` to see if it's still eligible to continue running. In this case it is, because we made no changes to it, so it will run again when it becomes the most eligible task.

The scheduler finishes its work, and is then called again because it's in an infinite `for()` loop. This time, because Salvo round-robins tasks of equal priority, the scheduler decides that `TaskShow()` is the most eligible task, and makes it run. First, `PORT` is configured as an output port and initialized.[29] Then TaskShow() enters its infinite loop for the first time, and 0x00 is written to `PORT` (the counter is now 0x0001), and once again `OS_Yield()` returns program execution to the scheduler after noting where to "return to" in `TaskShow()`. `TaskShow()` also remains eligible to run again.

After finishing its work, the scheduler is now called for the third time. Once again, `TaskCount()` is the most eligible task, and so it

---

27 Strictly speaking, this initialization is unnecessary, as all ANSI compilers will set `counter` to 0 before `main()`.

28 Because it was started first, and both tasks have the same priority.

29 In this example, each pin on I/O port `PORT` can be configured as an input or as an output. At power-up, all pins are configured as inputs, hence the need to configure them as outputs via `InitPORT()`.

runs again. But this time it resumes execution where we last left it, i.e. at the end of the `for()` loop. Since it's an infinite loop, execution resumes at the top of the loop. `TaskCount()` increments the counter, and relinquishes control back to the scheduler.

The next time the scheduler is called, `TaskShow()` resumes where it left off, goes to the top of its `for()` loop, writes to `PORT`, and yields back to the scheduler. This entire process of resuming a task where it left off, running the task, and returning control back to the scheduler is repeated indefinitely, with each task running alternately with every call to the scheduler.

When the program in Listing 25 runs, it gives the appearance of two separate things occurring simultaneously. Both tasks are free-running, i.e. the faster the processor, the faster they'll run. A counter appears to be incremented and sent to a port simultaneously. Yet we know that two separate tasks are involved, so we refer to this program as a multitasking application. It's not very powerful yet, and its functionality could be duplicated in many other ways. But as we add to this application we'll see that using Salvo will allow us to manage an increasingly sophisticated system with a minimal coding effort, and we'll be able to maximize the system's performance, too.

Using Events for Better Performance

The previous example did not use one of an RTOS' most powerful tools – intertask communications. It's also wasting processing power, since `TaskShow()` runs continuously, but `PORT` changes only once in every 512 calls to `TaskCount()`. Let's use intertask communication to make more efficient use of our processing power.

Listing 26 is shown below. We've used some `#define` preprocessor directives to improve legibility. This program is located in `salvo\tut\tu4\main.c`.

```
#include "main.h"
#include <salvo.h>

#define TASK_COUNT_P      OSTCBP(1) /* task #1 */
#define TASK_SHOW_P       OSTCBP(2) /* task #2 */
#define PRIO_COUNT        10 /* task priorities*/
#define PRIO_SHOW         10 /*  ""            */
#define SEM_UPDATE_PORT_P OSECBP(1) /* sem #1  */

_OSLabel(TaskCount1)
```

```
_OSLabel(TaskShow1)

unsigned int counter;

void TaskCount( void )
{
  for (;;) {
    counter++;

    if ( !(counter & 0x01FF) )
      OSSignalSem(SEM_UPDATE_PORT_P);

    OS_Yield(TaskCount1);
  }
}

void TaskShow( void )
{
  InitPORT();
  PORT = 0x00;

  for (;;) {
    OS_WaitSem(SEM_UPDATE_PORT_P,
      OSNO_TIMEOUT, TaskShow1);

    PORT = (PORT & ~0xFE)|((counter >> 8) & 0xFE);
  }
}

int main( void )
{
  Init();

  OSInit();

  OSCreateTask(TaskCount,
    TASK_COUNT_P, PRIO_COUNT);
  OSCreateTask(TaskShow,
    TASK_SHOW_P,  PRIO_SHOW);

  OSCreateSem(SEM_UPDATE_PORT_P, 0);

  counter = 0;

  for (;;)
    OSSched();
}
```

Listing 26: Multitasking with an Event

In Listing 26 we communicate between two tasks in order to update the port only when an update is required. We'll use a semaphore to represent this event. We initialize it to 0, meaning the event has not yet occurred. `TaskCount()` signals the semaphore whenever the upper 7 bits of the counter change. `TaskShow()`

waits for the event to occur, and then copies the upper 7 bits of the counter to `PORT`.

OSCreateSem()

`OSCreateSem()` creates a semaphore with the specified ecb pointer and initial value. A semaphore is created without any tasks waiting for it. A semaphore must be created before it can be signaled or waited.

OSSignalSem()

A semaphore is signaled via `OSSignalSem()`. If no task is waiting the semaphore, then it is simply incremented. If one or more tasks are waiting the semaphore, then the highest-priority waiting task is made eligible after incrementing the semaphore.

OS_WaitSem()

A task will wait a semaphore until the semaphore is signaled. If the semaphore is zero when the tasks waits it, then the task switches to the waiting state and returns through the scheduler. It will keep waiting for the semaphore until the semaphore is signaled and the task is the highest-priority task waiting for the semaphore. That's because more than one task can wait for a particular event.

If, on the other hand, the semaphore is nonzero when the task waits it, then the semaphore is decremented and the task continues its execution without context switching.

Tip The "`OS_`" prefix in `OS_WaitSem()` should remind you that a context switch may occur in a call to `OS_WaitSem()`, depending on the value of the semaphore.

Tip You must always specify a timeout[30] when waiting a semaphore via `OS_WaitSem()`. If you want the task to wait forever for the semaphore to be signaled, use the predefined value `OSNO_TIMEOUT`.

Note In this example, `OS_WaitSem()` is used in place of `OS_Yield()`. In fact, the macro `OS_WaitSem()` includes a call to `OS_Yield()`. You do not need to call `OS_Yield()` when using a conditional context switcher like `OS_WaitSem()` – it does it for you.

In Depth

In order to improve the performance of our application, we'd like to update `PORT` only when the counter's upper 7 bits change. To do

[30] The timeout parameter is required regardless of whether or not your application is built with Salvo code (source files or libraries) that supports timeouts. This makes it possible to rebuild applications for timeouts without any user source code changes.

this we will use a signaling mechanism between the two tasks, called a semaphore. Here, the semaphore is a flag that's initialized to zero to mean that there's no need to update the port. When the semaphore is signaled, i.e. it is made non-zero, it means that a `PORT` update is required.

Inter-task communication is achieved by using the semaphore to alert the waiting task (in this case, `TaskShow()`) that a `PORT` update is required. This is done in `TaskCount()` by calling `OSSignalSem()` with the parameter being the semaphore, and by having `TaskShow()` wait the semaphore.

Note `TaskCount()` does not know which task(s) is(are) waiting on the semaphore, and `TaskShow()` does not know how the semaphore is signaled.

The first time `TaskShow()` runs through the scheduler it calls `OS_WaitSem()`. Since the semaphore was initialized to zero, `TaskShow()` yields control back to the scheduler and changes its state from eligible to waiting. Now there is only one eligible task, `TaskCount()`, and the scheduler runs it repeatedly.

When `TaskCount()` finally signals the semaphore, `TaskShow()` is made eligible again and will run once `TaskCount()` returns through the scheduler. After all, since the counter's upper 7 bits change only every 512 calls to `TaskCount()`, there's no point in running it more often than that. By using a semaphore, `TaskShow()` runs only when it needs to update `PORT`. The rest of the time, it is waiting and does not consume any processing power (instruction cycles).

The performance of this application is roughly twice as good (i.e. the counter increments at twice the speed) as that of Listing 25. That's because a waiting task consumes no processor power whatsoever while it waits – recall that the scheduler only runs tasks that are eligible. Since `TaskShow()` is waiting for the semaphore over 97% of the time,[31] it runs only on the rare occasion that the counter's upper byte has changed. The rest of the time, the scheduler is running `TaskCount()`.

It should be apparent that the calls to `OS_WaitSem()` and `OSSignalSem()` above implement some powerful functionality. In this example, these Salvo event services control when `TaskShow()` will run by using a semaphore for intertask communications. Here

[31] Measured on Test System A.

the semaphore is a simple flag (1 bit of information). Salvo supports the use of semaphores and other mechanisms to pass more information (e.g. a count, or a pointer) from one task to another.

Listing 26 is a complete Salvo program – nothing is missing. There's nothing "running in the background", nothing checking to see if a waiting task should be made eligible, etc. In other words, there's no polling going on – all of Salvo's actions are event-driven, which contributes to its high performance. `TaskShow()` goes from waiting to eligible in the call to `OSSignalSem()`, and from running to waiting via `OS_WaitSem()`. With Salvo, you have complete control over what the processor is doing at any one time, and so you can optimize your program's performance without unwanted interference from the RTOS.

Delaying a Task

One thing missing from the previous example is any notion of real-time performance. If we add other tasks of equal or higher priority to the application, the rate at which the counter increments will decline. Let's look at how an RTOS can provide real-time performance by adding a task that runs at 2Hz, regardless of what the rest of the system is doing. We'll do this by repetitively delaying a task.

Being able to delay a task for a specified time period can be a very useful feature. A task will remain in the delayed state, ineligible to run, until the delay time specified has expired. It's up to the kernel to monitor delays and return a delayed task to the eligible state.

The application in Listing 27 blinks the LED on the least significant bit of `PORT` at 1Hz by creating and running a task which delays itself 500ms after toggling the port bit, and does this repeatedly. This program is located in `salvo\tut\tu5\main.c`.

```
#include "main.h"
#include <salvo.h>

#define TASK_COUNT_P      OSTCBP(1)  /* task #1 */
#define TASK_SHOW_P       OSTCBP(2)  /*  ""  #2 */
#define TASK_BLINK_P      OSTCBP(3)  /*  ""  #3 */
#define PRIO_COUNT        10  /* task priorities*/
#define PRIO_SHOW         10  /*  ""            */
#define PRIO_BLINK        2   /*  ""            */
#define SEM_UPDATE_PORT_P OSECBP(1)  /* sem #1  */

unsigned int counter;

_OSLabel(TaskCount1)
```

```
_OSLabel(TaskShow1)
_OSLabel(TaskBlink1)

void TaskCount( void )
{
  for (;;) {
    counter++;

    if ( !(counter & 0x01FF) )
      OSSignalSem(SEM_UPDATE_PORT_P);

    OS_Yield(TaskCount1);
  }
}

void TaskShow( void )
{
  for (;;) {
    OS_WaitSem(SEM_UPDATE_PORT_P,
      OSNO_TIMEOUT, TaskShow1);

    PORT = (PORT & ~0xFE)|((counter >> 8) & 0xFE);
  }
}

void TaskBlink( void )
{
  InitPORT();
  PORT = 0x00;

  for (;;) {
    PORT ^= 0x01;

    OS_Delay(50, TaskBlink1);
  }
}

void main( void )
{
  Init();

  OSInit();

  OSCreateTask(TaskCount,
    TASK_COUNT_P, PRIO_COUNT);
  OSCreateTask(TaskShow,
    TASK_SHOW_P,  PRIO_SHOW);
  OSCreateTask(TaskBlink,
    TASK_BLINK_P, PRIO_BLINK);

  OSCreateSem(SEM_UPDATE_PORT_P, 0);

  counter = 0;

  OSEi();

  for (;;)
```

```
    OSSched();
}
```

Listing 27: Multitasking with a Delay

Additionally, interrupts are required to call `OSTimer()` at the desired system tick rate of 100Hz. The code to do this is located in `salvo\tut\tu1\sysa\isr.c`:[32]

```
#include "salvo.h"

#define TMR0_RELOAD 156 /* for 100Hz ints @ 4MHz
*/

void interrupt IntVector( void )
{
    if ( T0IE && T0IF ) {
        T0IF = 0;
        TMR0 -= TMR0_RELOAD;

        OSTimer();
    }
}
```

Listing 28: Calling OSTimer() at the System Tick Rate

In order to use delays in a Salvo application, you must add the Salvo system timer to it. In the above example we've added a 10ms system timer by calling `OSTimer()` at a periodic rate of approximately 100Hz. The periodic rate is derived by a timer overflow, which causes an interrupt. Interrupts must be enabled in order for `OSTimer()` to be called – hence the call to `OSEi()` just prior to starting multitasking. Since delays are specified in units of the system tick rate, the blink task is delayed by 50*10ms, or 500ms.

OSTimer()

In order to use Salvo delay services, you must call `OSTimer()` at a regular rate. This is usually done with a periodic interrupt. The rate at which your application calls `OSTimer()` will determine the resolution of delays. If the periodic interrupt occurs every 10ms, by calling `OSTimer()` from within the ISR you will have a system tick period of 10ms, or a rate of 100Hz. With a tick rate defined, you can specify delays to a resolution of one timer tick period, e.g. delays of 10ms, 20ms, ... 1s, 2s, ... are possible.

Note Salvo's timer features are highly configurable, with delays of up to 32 bits of system ticks, and with an optional prescalar. Consult *Chapter 5 • Configuration* and *Chapter 6 • Frequently Asked Questions (FAQ)* for more information.

[32] `IntVector()` is also used in `tu6`, below. `IntVector()` (and hence the contents of `isr.c`) are target- and compiler-specific.

OS_Delay()

With `OSTimer()` in place and called repetitively at the system tick rate, you can now delay a task by replacing `OS_Yield()` with a call to `OS_Delay()`, which will force the context switch and delay the task for the number of system ticks specified. The task will automatically become eligible once the specified delay has expired.

In Depth

In Listing 27, each time `TaskBlink()` runs, it delays itself by 500ms and enters the delayed state upon returning to the scheduler. When `TaskBlink()`'s delay expires 500ms later it is automatically made eligible again, and will run after the current (running) task context-switches. That's because `TaskBlink()` has a higher priority than either `TaskCount()` or `TaskShow()`. By making `TaskBlink()` the highest-priority task in our application, we are guaranteed a minimum of delay (latency) between the expiration of the delay timer and when `TaskBlink()` toggles bit 0 of `PORT`. Therefore `TaskBlink()` will run every 500ms with minimal latency, irrespective of what the other tasks are doing.

Tip If `TaskBlink()` had the same priority as `TaskCount()` and `TaskShow()`, it would occasionally remain eligible (and would not run) while both `TaskCount()` and `TaskShow()` ran before it. Its maximum latency would increase. If `TaskBlink()` had a lower priority, it would never run at all.

The initialization of `PORT` was moved to `TaskBlink()` because of `TaskBlink()`'s priority. It will be the first task to run, and therefore `PORT` will be initialized as an output before `TaskShow()` runs for the first time.

Salvo monitors delayed tasks once per call to `OSTimer()`, and the overhead is independent of the number of delayed tasks.[33]

This illustrates that the system timer is useful for a variety of reasons. A single processor resource (e.g. a periodic interrupt) can be used in conjunction with `OSTimer()` to delay an unlimited number of tasks. More importantly, delayed tasks consume only a very small amount of processing power while they are delayed, much less than running tasks.

[33] Except when one or more task delays expire simultaneously.

Signaling from Multiple Tasks

A multitasking approach to programming delivers real benefits when priorities are put to good use and program functionality is clearly delineated along task lines.

Review the code in Listing 29 to see what happens when we lower the priority of the always-running task, `TaskCount()`, and have `TaskShow()` handle all writes to `PORT`. This program is located in `salvo\tut\tu6\main.c`.

```
#include "main.h"
#include <salvo.h>

#define TASK_COUNT_P      OSTCBP(1)  /* task #1 */
#define TASK_SHOW_P       OSTCBP(2)  /*  ""  #2 */
#define TASK_BLINK_P      OSTCBP(3)  /*  ""  #3 */
#define PRIO_COUNT        12  /* task priorities*/
#define PRIO_SHOW         10  /*  ""            */
#define PRIO_BLINK        2   /*  ""            */
#define MSG_UPDATE_PORT_P OSECBP(1)  /* sem #1  */

unsigned int counter;

char CODE_B = 'B';
char CODE_C = 'C';

_OSLabel(TaskCount1)
_OSLabel(TaskShow1)
_OSLabel(TaskBlink1)
_OSLabel(TaskBlink2)

void TaskCount( void )
{
  counter = 0;

  for (;;) {
    counter++;

    if ( !(counter & 0x01FF) )
      OSSignalMsg(MSG_UPDATE_PORT_P,
        (OStypeMsgP) &CODE_C);

    OS_Yield(TaskCount1);
  }
}

void TaskShow( void )
{
  OStypeMsgP msgP;

  InitPORT();
  PORT = 0x00;

  for (;;) {
```

```
    OS_WaitMsg(MSG_UPDATE_PORT_P, &msgP,
      OSNO_TIMEOUT, TaskShow1);

    if ( *(char *)msgP == CODE_C ) {
      PORT = (PORT & ~0xFE)|((counter >> 8)&0xFE);
    }
    else {
      PORT ^= 0x01;
    }
  }
}

void TaskBlink( void )
{
  OStypeErr err;

  for (;;) {
    OS_Delay(50, TaskBlink1);

    err = OSSignalMsg(MSG_UPDATE_PORT_P,
      (OStypeMsgP) &CODE_B);

    if ( err == OSERR_EVENT_FULL ) {
      OS_SetPrio(PRIO_SHOW+1, TaskBlink2);
      OSSignalMsg(MSG_UPDATE_PORT_P,
       (OStypeMsgP) &CODE_B);
      OSSetPrio(PRIO_BLINK);
    }
  }
}

void main( void )
{
  Init();

  OSInit();

  OSCreateTask(TaskCount,
    TASK_COUNT_P, PRIO_COUNT);
  OSCreateTask(TaskShow,
    TASK_SHOW_P,  PRIO_SHOW);
  OSCreateTask(TaskBlink,
    TASK_BLINK,   PRIO_BLINK);

  OSCreateMsg(MSG_UPDATE_PORT_P, (OStypeMsgP) 0);

  OSEi();

  for (;;)
    OSSched();
}
```

Listing 29: Signaling from Multiple Tasks

In Listing 29 we've made two changes to the previous program. First, `TaskShow()` now handles all writes to `PORT`. Both `Task-`

`Count()` and `TaskBlink()` send a unique message to `TaskShow()` (the character ‘C' for "count" or ‘B' for "blink", respectively) which it then interprets to either show the counter on the port or toggle the least significant bit of the port. Second, we've lowered the priority of `TaskCount()` by creating it with a lower priority.

OSCreateMsg()

`OSCreateMsg()` is used to initialize a message. Salvo has a defined type for messages, and requires that you initialize the message properly. A message is created without any tasks waiting for it. A message must be created before it can be signaled or waited.

Note Salvo services require that you interface your code using predefined types, e.g. `OStypeMsgP` for message pointers. You should use Salvo's predefined types wherever possible. See *Chapter 7 • Reference* for more information on Salvo's predefined types.

OSSignalMsg()

In order to signal a message with `OSSignalMsg()`, you must specify both a ecb pointer and a pointer to the message contents. If no task is waiting the message, then the message gets the pointer, unless the message is already defined, in which case an error has occurred. If one or more tasks are waiting the message, then the highest-priority waiting task is made eligible. You must correctly typecast the message pointer so that it can be dereferenced properly by whichever tasks wait the message.

OS_WaitMsg()

A task waits a message via `OS_WaitMsg()`. The message is returned to the task through a message pointer. In order to extract the contents of the message, you must dereference the pointer with a typecast matching what the message pointer is pointing to.

OS_SetPrio()

A task can change its priority and context-switch immediately thereafter using `OS_SetPrio()`.

OSSetPrio()

A task can change its priority using `OSSetPrio()`. The new priority will take effect as soon as the task yields to the scheduler.

In Depth

`TaskShow()` is now the only task writing to `PORT`. A single message is all that is required to pass unique information from two different tasks (which run at entirely different rates) to `TaskShow()`. In this case, the message is a pointer to a 1-byte constant. Since messages contain pointers, casting and proper dereferencing are required to send and receive the intended information in the message.

InListing 29, the following scenario is possible: Immediately after `TaskCount()` signals the message, `TaskBlink()`'s delay expires and `TaskBlink()` is made eligible to run. Since `TaskBlink()` has the highest priority, the message will still be present when `TaskBlink()` signals the message. Therefore `OSSignalMsg()` will return an error. The LED's PORT pin will fail to toggle …

This example illustrates the use of *return values* for Salvo services. By testing for the abovementioned error condition, we can guarantee the proper results by temporarily lowering `TaskBlink()`'s priority and yielding to the scheduler before signaling the message again. `TaskShow()` will temporarily be the highest-priority task, and it will "claim" the message. As long as `TaskCount()` does not signal messages faster than once every three context switches, this solution remains a robust one.[34]

In a more sophisticated application, e.g. a car's electronics, one can imagine `TaskShow()` being replaced with a task that drives a dashboard display divided into distinct regions. Four tasks would monitor information (e.g. rpm, speed, oil pressure and water temperature) and would pass it on by signaling a message whenever a parameter changed. `TaskShow()` would wait for this message. Each message would indicate where to display the parameter, what color(s) to use (e.g. red on overtemperature) and the parameter's new value. Since visual displays generally have low refresh rates, `TaskShow()` could run at a lower priority than the sending tasks. These tasks would run at higher priority so as to process the information they are sampling without undue interference from the slow display task. For example, the oil-pressure-monitoring task might run at the highest priority, since a loss of oil pressure means certain engine destruction. By having the display functionality in a task instead of in a callable function, you can fine-tune the performance of your program by assigning an appropriate priority to each of the tasks involved.

By lowering `TaskCount()`'s priority we've changed the behavior of our application. `PORT` updates now take precedence over the counter incrementing. This means that `PORT` updates will occur sooner after the message is signaled. The counter now increments only when there's nothing else to do. You can dramatically and predictably alter the behavior of your program by changing just the priority when creating a task.

---

34 An alternative solution to this problem would be to use a message queue with room for two messages in it.

Wrapping Up

As a Salvo user you do not have to worry about scheduling, tasks states, event management or intertask communication. Salvo handles all of that for you automatically and efficiently. You need only create and use the tasks and events in the proper manner to get all of this functionality, and more.

Note *Chapter 7 • Reference* contains working examples with commented C source code for every Salvo user service. Refer to them for more information on how to use tasks and events.

Food For Thought

Now that you're writing code with task- and event-based structures like the ones Salvo provides, you may find it useful or even necessary to change the way you approach new programs. Instead of worrying about how many processor resources, ISRs, global variables and clock cycles your application will require, focus instead on the tasks at hand, their priorities and purposes, your application's timing requirements and what events drive its overall behavior. Then put it all together with properly prioritized tasks that use events to control their execution and to communicate inside your program.

Part 2: Compiling a Salvo Application

Note If you have not done so already, please follow the instructions in *Chapter 3 • Installation* to install all of Salvo's components onto your computer. You may also find it useful to refer to *Chapter 5 • Configuration* and *Chapter 7 • Reference* for more information on some of the topics mentioned below

Now that you are familiar with how to write a Salvo application, let's compile it into an executable program. Below you will find general instructions on compiling a Salvo program, followed by a step-by-step procedure to compile and test one of the tutorial programs.

Working Environment

Salvo is distributed as a collection of source code files, object files, library files and other support files. Since all source code is provided in the full version, Salvo can be compiled on many devel-

opment platforms. You will need to be proficient with your editor / compiler / integrated development environment (IDE) in order to successfully compile a Salvo application.

You should be familiar with the concepts of including a file inside another file, compiling a file, linking one or more files, working with libraries, creating an executable program, viewing the debugging output of your compiler, and placing your program into memory.

Please refer to your editor's / compiler's / IDE's documentation on how to include files into source code, compile source code, link to separate object modules, and compile and link to libraries.

Many IDEs support an automatic make-type utility. You will probably find this very useful when working with Salvo. If you do not have a make utility, you may want to investigate obtaining one. Both commercial and freeware / shareware make utilities exist, for command-line hosts (e.g. DOS) and Windows 95 / 98 / 2000 / NT.

Creating a Project Directory

In creating an application with Salvo you'll include Salvo source files in your own source code, and you'll probably also link to Salvo object files or Salvo libraries. We strongly recommend that you do not modify any Salvo files directly, nor should you duplicate any Salvo files unnecessarily. Unless you intend to make changes to the Salvo source code, you should consider all of Salvo's files as being read-only.

By creating a working directory for each new Salvo application you write, you'll be able to:

- minimize hard disk usage,
- manage your files better,
- make changes to one application without affecting any others, and
- compile unique versions of Salvo libraries for different projects.

Note Complete projects for all the tutorial programs can be found in `salvo\tut\tu1-tu6`.

Including salvo.h

Salvo's main header file, `salvo.h`, must be included in each of your source files that uses Salvo. You can do this by inserting

```
#include <salvo.h>
```

or

```
#include "salvo.h"
```

into your source files, depending on how you've set up your compiler to search for included files. If you include a project header file (e.g. `myproject.h`) in all of your source files, you may want to add one of the lines above to it.

`salvo.h` contains the line

```
#include "salvocfg.h"
```

to automatically include your project-specific version of `salvocfg.h` (see *Setting Configuration Options*, below). You should not include `salvocfg.h` in any of your source files – just including `salvo.h` is enough.

Note `salvo.h` has a built-in "include guard" which will prevent problems when multiple references to include `salvo.h` are contained in a single source file.

Configuring your Compiler

In order to successfully compile your Salvo application you must configure your compiler for use with the Salvo source files and libraries. You have several options available to you when combining your code with the Salvo source code in order to build an application.

Setting Search Paths

First, you must specify the appropriate search paths so that the compiler can find the necessary Salvo include (`*.h`) and source (`*.c`) files.

Tip If your compiler does not support search paths, you can copy Salvo files from their source directories to your project directory

and have the compiler find it by virtue of the fact that it's in the current directory. However, this is not recommended for obvious reasons.

At the very least, your compiler will need to know where to find the following files:

- `salvo.h`, located in `salvo\inc`
- `salvocfg.h`, located in your current project directory

You may also need to specify the Salvo source file directory (`salvo\src`) if you plan to include Salvo source files in your own source files (see below).

At this point you'll need to decide how you will incorporate Salvo into your application. A decision is required because how you do this may greatly affect the size of your application. If you're developing an application which runs on a processor with lots of ROM and RAM, then you may choose to either include salvo files directly into your source files, or compile and link to them as object files. However, if your processor has very limited ROM and RAM, you'll probably want to make as much use of the libraries as possible to minimize the amount of code (ROM) and data (RAM) that Salvo adds to your application. That's because linkers extract only those library functions needed for the final executable, thus minimizing the size of your application.

Using Libraries vs. Using Source Files

Different methods for incorporating Salvo into your application are outlined below. Linking to Salvo libraries is the simplest method, but has limitations. Including the Salvo source files in your project is the most flexible method, but isn't as simple. Creating custom Salvo libraries from the source files is for advanced users.

Using Libraries

Just like a C compiler's library functions – e.g. `rand()` in the standard library (`stdlib.h`) or `printf()` in the standard I/O library (`stdio.h`) – Salvo has functions (called *user services*) contained in libraries. Unlike a compiler's library functions, Salvo's user services are highly configurable – i.e. their behavior can be controlled based on the functionality you desire in your application. Each Salvo library contains user functions compiled for a particular set of *configuration options*. There are many different Salvo libraries.

Note Configuration options are *compile-time* tools used to configure Salvo's source code and generate libraries. Therefore the functionality of a precompiled library *cannot be changed* through configuration options. To change a library's functionality, it must be regenerated (i.e. re-compiled) with new configuration options.

In order to facilitate getting started, all Salvo distributions contain libraries with most of Salvo's functionality already included. As a beginner, you should start by using the libraries to build your applications. This way, you don't have to concern yourself with the myriad of configuration options. Once you need a particular functionality that's only enabled through non-default configuration options, then you should use the configuration options in conjunction with the Salvo source code.

Note The sections below pertain primarily to projects that use the source files contained in the full Salvo distribution. If you are using the freeware or standard libraries, you should refer to CHAPTER 8 Libraries for information on setting the configuration options.

Whichever method you choose, you should read and, below, to understand the purpose of Salvo's configuration options and how they are used.

Tip The easiest and quickest way to create a working application is to link your source code to the appropriate Salvo library. **AN-1 Using Salvo Freeware Libraries** *with the HI-TECH PICC Compiler* and **AN-4 Building a Salvo Application** *with HI-TECH PICC and Microchip MPLAB* describe in detail how to create applications with Salvo libraries.

Linking to Salvo Libraries

Accessing Salvo's features through a library is often the best solution. You can either use the *freeware* or *standard* libraries supplied with Salvo, or you can create your own *custom libraries*, based on your project's configuration file, `salvocfg.h`. Using libraries will usually result in the smallest possible executable, because your compiler will extract only those library functions used by your application.

Note In order to create and use custom Salvo libraries you'll need to be familiar with creating and using libraries with your compiler and/or in your development environment. Consult their documentation for more information.

Tip You may find that using a make utility (built into your development environment or stand-alone) will greatly simplify the process of creating your own applications and/or custom libraries.

Salvo provides freeware and standard libraries for certain supported processors and compilers in `salvo\lib`. While they are a good starting point for creating a Salvo application, certain factors may entice you to create and use your own libraries instead. Among them are:

- the number of tasks and events that the libraries support[35] may not suit your application,
- the libraries may not be optimized (size- or speed-wise) for the demands of your application, and
- the allocation of RAM for Salvo's variables (through the `OSLOC_XYZ` configuration options) may not suit your application.

For these and other reasons, after you've successfully creating a working Salvo application that uses a library you may wish to consider creating custom libraries for your application(s).

Tip When creating an application with two or more libraries that contain a function with the same name, your application will (usually) end up using the *first* such function found by the linker. Therefore the order in which you specify libraries is important.

Note When linking your application to precompiled Salvo libraries (freeware, standard or custom), remember that they were created with a potentially different set of configuration options than are currently in your `salvocfg.h`. For this reason, you must recreate libraries used by your application whenever you change the application's `salvocfg.h`. In other words, your application must use *exactly* the same configuration options as those found in the `salvocfg.h` file used to create the associated library. Otherwise the Salvo features your application attempts to use may not be available, and you may encounter errors while compiling, linking or running your application.

35 These (and other parts of Salvo) are fixed at compile time.

Building the Tutorial Program

Please refer to *AN-4 Building a Salvo Application with HI-TECH PICC and Microchip MPLAB* for a complete description of how to build the tutorial program using freeware or standard libraries.

Complete library-based MPLAB projects for all the tutorial programs can be found in `salvo\tut\tu1-tu6`. See *Appendix C • File and Program Descriptions* for more information.

Using Source Files

Salvo is configurable primarily to minimize the size of the user services and thus conserve ROM. Also, its configurability aids in minimizing RAM usage. Without it, Salvo's user services and variables might be too large to be of any use in many application. All of this has its advantages and disadvantages – on the one hand, you can fine-tune Salvo to use just the right amount of ROM and RAM in your application. On the other hand, it can be a challenge learning how all the different configuration options work.

There are some instances where it's better to create your application by adding the Salvo source files as nodes to your project. When you use this method, you can change configuration options and re-build the application to have those changes take effect in the Salvo source code. The rest of this chapter covers this approach.

Setting Configuration Options

Salvo is highly configurable. You'll need to create and use a configuration file, `salvocfg.h`, for each new application you write. This simple text file is used to select Salvo's compile-time configuration options, which affect things like how many tasks and events your application can use. All configuration options have default values – most of them may be acceptable to your application.

Note Whenever you redefine a configuration option in `salvocfg.h`, you *must* recompile all of the Salvo source files in your application.

One way to set configuration options is to copy the default `salvocfg.h` file (`salvo\inc\user\salvocfg.h`) to your project directory and edit it there. However, because there are so many configuration options, we recommend instead that you build a configuration file from a new, empty file. `salvocfg.h` should be

placed in your project directory so that your compiler will find it when compiling your source code.

The examples below assume that you are creating and editing `salvocfg.h` via a text editor. Each configuration option is set via a C-language `#define` statement. For example, to configure Salvo to support 16-bit delays, you would add

```
#define OSBYTES_OF_DELAYS 2
```

to your project's `salvocfg.h` file.

Note The name and value of the configuration option are case-sensitive. If you type the name incorrectly, the option will be overridden by the Salvo default.

Identifying the Compiler

You will need to identify the compiler you will be using to create your application. If your compiler is already supported, choose it from the list of supported compilers. For example, if you were using the HI-TECH PICC compiler, you'd add

```
#define OSCOMPILER OSHT_PICC
```

to `salvocfg.h`. If your compiler is not supported, choose a value of `OSUNKNOWN` or don't define it at all. You'll need to refer to *Chapter 10 • Porting* for more information on working with as-yet-unsupported compilers.

Identifying the Target Processor

Next, you must identify the processor used in your application. This may happen if the compiler you're using supports fundamentally different types or families of processors. In this case you'll need to choose it from the list of supported processors, e.g.

```
#define OSTARGET OSPIC16
```

If your processor is not supported, choose a value of `OSUNKNOWN` or don't define it at all. If your target processor requires special needs, you'll need to refer to *Chapter 10 • Porting* for more information on working with as-yet-unsupported processors.

Note The `OSTARGET` configuration option identifies the processor that will be running Salvo code, not the processor that runs your compiler.

Specifying the Number of Tasks

Memory for Salvo's internal task structures is allocated at compile time. You must specify in `salvocfg.h` how many tasks you would like supported in your application, e.g.:

```
#define OSTASKS 4
```

You do not need to use all the tasks that you allocate memory for, nor must you use their respective tcb pointers (numbered from `OSTCBP(1)` to `OSTCBP(OSTASKS)`) consecutively. If you attempt to reference a task for which no memory was allocated, the Salvo user service will return a warning code.

Tip Tasks are referred to in Salvo by their tcb pointers. It's recommended that you use descriptive designations in your code to refer to your tasks. This is most easily done by using the `#define` statement in your project's main header (`.h`) file, e.g.:

```
#define TASK_CHECK_TEMP_P[36] OSTCBP(1)
#define TASK_MEAS_SPEED_P  OSTCBP(2)
#define TASK_DISP_RPM_P    OSTCBP(3)
```

Your program will be easier to understand when calling Salvo task services with meaningful names like these.

Specifying the Number of Events

Memory for Salvo's internal event structures is also allocated at compile time. You must specify in `salvocfg.h` how many events you would like supported in your application, e.g.:

```
#define OSEVENTS 3
```

Events include semaphores (binary and counting), messages and message queues.

You do not need to use all the events that you allocate memory for, nor must you use their respective ecb pointers (numbered from

[36] The `P` suffix is there to remind you that the object is a Pointer to something.

OSECBP(1) to OSECBP(OSEVENTS)) consecutively. If you attempt to reference an event for which no memory was allocated, the Salvo user service will return a warning code.

If your application does not use events, leave OSEVENTS undefined in your salvocfg.h, or set it to 0.

Tip You should use descriptive names for events, too. See the tip above on how to do this.

Specifying other Configuration Options

You may also need to specify other configuration options, depending on which of Salvo's features you plan to use in your application. Many of Salvo's features are not available until they are enabled via a configuration option. This is done to minimize the size of the code that Salvo adds to your application. For small projects, a small salvocfg.h may be adequate. For larger projects and more complex applications, you will need to select the appropriate configuration option(s) for all the features you wish to use. Other configuration options include:

- the size of delays, counters, etc. in bytes,
- the size of semaphores and message pointers, and
- memory-locating directives specific to the compiler.

Tip If you attempt to use a Salvo feature by calling a Salvo function and your compiler issues an error message suggesting that it can't find the function, this may be because the function has not been enabled via a configuration option.

In a sophisticated application, some of the additional configuration options might be:

```
#define OSBYTES_OF_DELAYS     3
#define OSTIMER_PRESCALAR     20
#define OSLOC_ECB             bank3
```

The values for the options will either be numeric constants, predefined constants (e.g. TRUE and FALSE), or definitions provided for the compiler in use (e.g. bank3, used by the HI-TECH PICC compiler to locate variables in a particular bank of memory).

Including Salvo Files

One way to get started is to have your project's `main.c` include the necessary Salvo source files via the C `#include` preprocessor directive. You can include just those Salvo files that contain the routines you're using, plus a few others like `mem.c`, which contains Salvo's data structures.

This approach may become unwieldy since your compiler has to preprocess and compile all of the Salvo source code each time you recompile your application. Your application will also contain all of the functionality of each one of those Salvo source code files, even if your application doesn't actually call each of the functions in those files.

This is also the slowest method, and does not take advantage of your compiler's make utility and project management, if it has them.

Note There are some instances where it is advantageous to include certain Salvo source code files in one of your source code files, instead of accessing the Salvo functions in those files through a library. This usually concerns the issue of context saving during interrupts, and by organizing the files in this way, unnecessary context saving and restoring can be eliminated. See *Chapter 11 • Tips, Tricks and Troubleshooting* for more details.

Linking to Salvo Object Files

You can create an application by compiling and then linking your application to some or all of Salvo's `*.c` source files. This method is recommended for most applications, and is compatible with make utilities. It is relatively straightforward, but has the disadvantage that your final executable may contain all of the Salvo functionality contained in the linked files, regardless of whether your application uses them or not.

Note Some compilers are capable of "smart linking" whereby functions that are linked but not used do not make it into the final executable. In this situation there is no downside to linking your application to all of Salvo's source files.

Chapter 7 • Reference contains descriptions of all the Salvo user services, and the Salvo source files that contain them. As soon as you use a service in your code, you'll also need to link to the ap-

propriate source file. This is usually done in the compiler's IDE by adding the Salvo source files to your project. If you use the service without adding the file, you will get a link error when you make your project.

The size of each compiled object module is highly dependent on the configuration options you choose. Also, you can judiciously choose which modules to compile and link to – for example, if don't plan on using dynamic task priorities in your application, you can leave out `prio.c`, for a reduction in code size.

Building the Tutorial Program

If you are using HI-TECH PICC with Microchip MPLAB, please refer to Application Note *AN-4 Building a Salvo Application with HI-TECH PICC and Microchip MPLAB*. It explains in detail how to create, build and test the tutorial project located in `salvo\tut\tu6` with HI-TECH's PICC compiler and Microchip's MPLAB IDE.

The rest of this chapter describes how to create a Salvo application using the HI-TECH PICC compiler with the HI-TECH HPDPIC IDE. The basic concepts (identifying `main.c`, setting include paths, adding project nodes) can be carried over to other compilers and development environments.

Getting Started

Note The project described below is contained in the HPDPIC project files `tu6.prj` and `tu6free.prj` located in `salvo\tut\tu6\sysa`. You may use it to verify the actions listed below. Alternately, you can create your own project under a different name and in a different directory.

Let's build an actual program and see it run. We'll compile and link the program shown in Listing 29, using Salvo's freeware libraries and using the full version's source code modules. The compiler is the HI-TECH PICC compiler. The target is Salvo Test System A, a Microchip PIC16C77 running on the Microchip PICDEM-2 demonstration board. We'll use the HI-TECH HPDPIC IDE to manage the project, and download the compiler-generated hex file to the target via the Microchip MPLAB IDE to a Microchip PICMASTER in-circuit emulator. Salvo and all of these tools are installed on a Wintel platform.

Creating the Project

Launch HPDPIC and create a new project:

> Make > New project ... > C:\salvo\tut\tu6\sysa\tu6.prj > OK

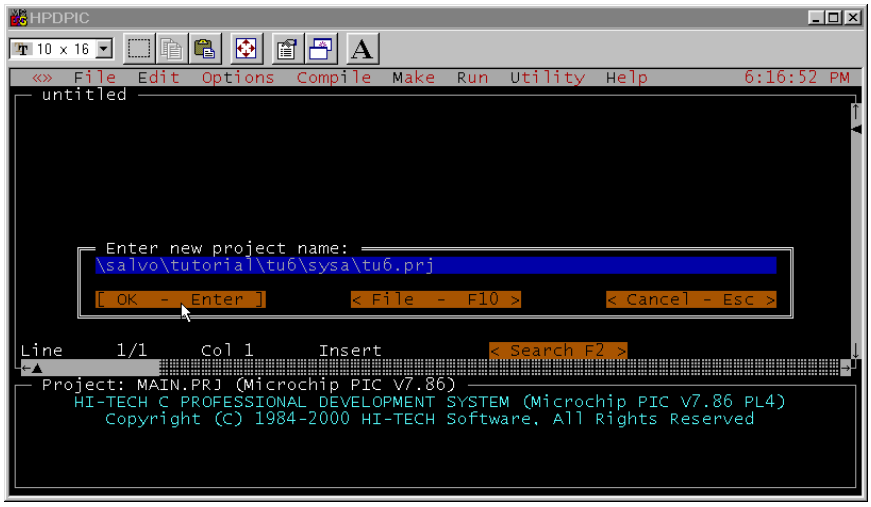


Figure 26: Creating the Tutorial Project

This automatically gives the project a name, `tu6`. Select the processor:

> Midrange > 16C77 > OK

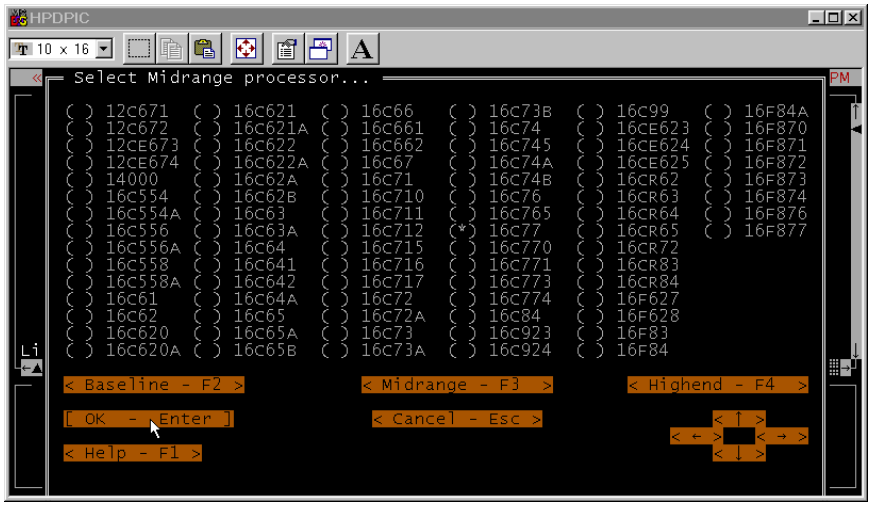


Figure 27: Selecting the Processor Type

Select the floating point type:[37]

37 This tutorial program does not use floating point routines.

> 24 bit double float > OK

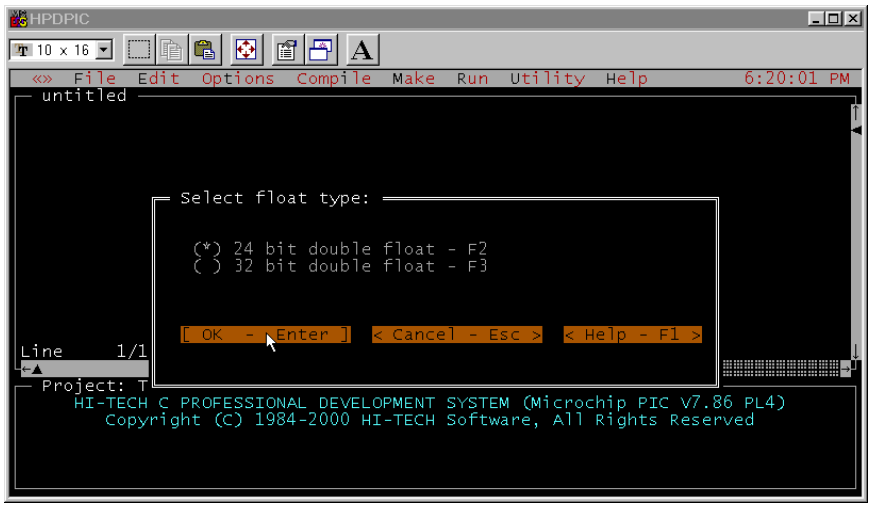


Figure 28: Selecting the Float Type

Select the output file format:

> Bytecraft .COD file > OK

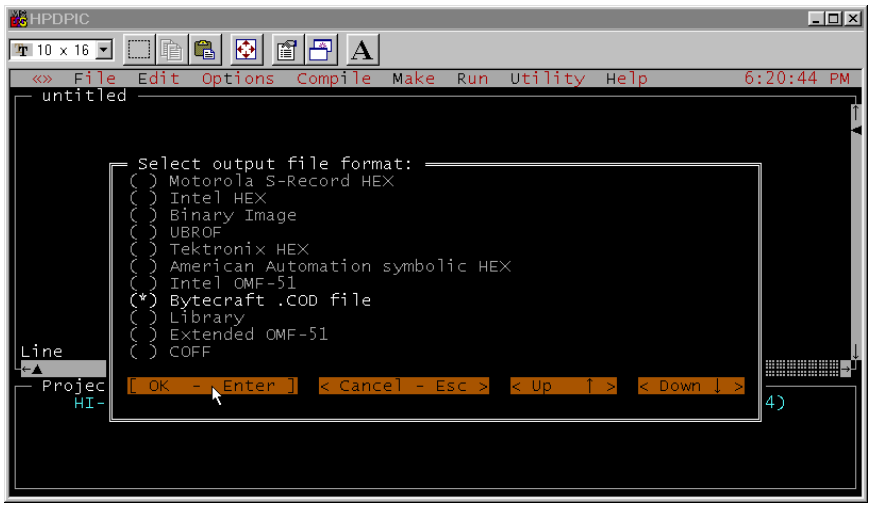


Figure 29: Selecting the Output File Format

Select the compiler optimizations:

> Full optimization > Global optimization level 5 > OK

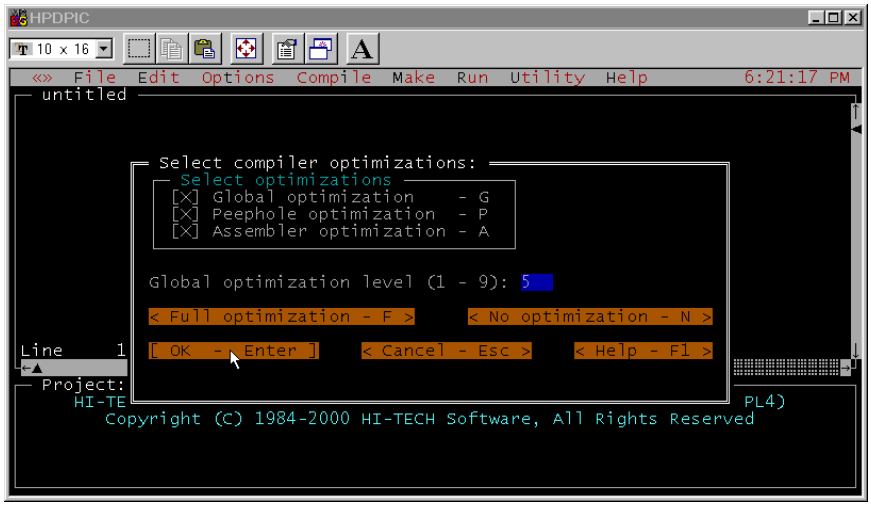


Figure 30: Selecting the Optimizations

Leave the map and symbol file options at their default values:

> OK

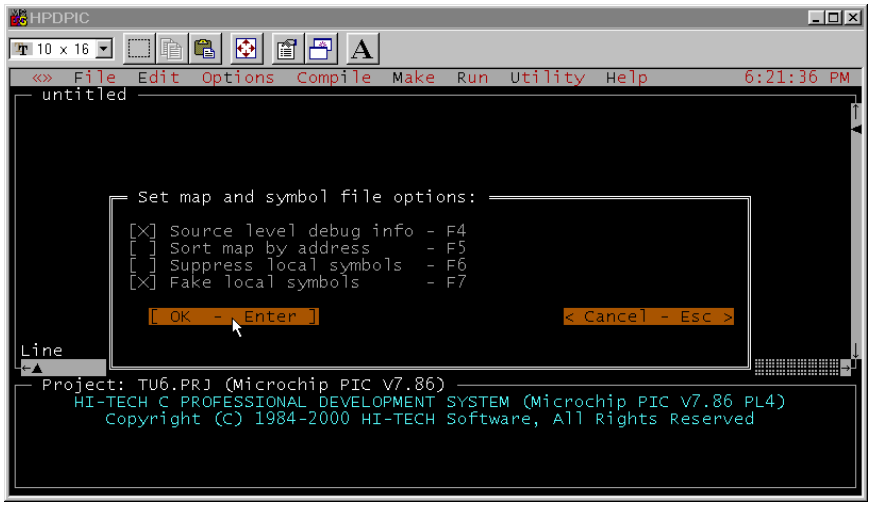


Figure 31: Selecting the Map and Symbol File Options

Add the source file containing the program in Listing 29. It is located in `C:\salvo\tut\tu6\main.c`. Add it to the project's source files:

Make > Source file list … > ..\main.c[38] > DONE

38 Wherever possible, projects contained in the Salvo distribution use relative pathnames to aid in portability.

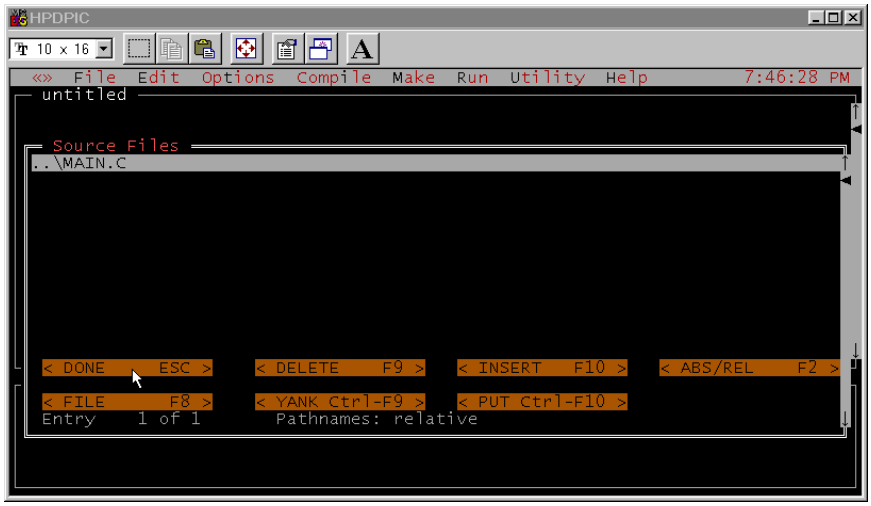


Figure 32: Adding the Source File main.c

Save the project:

> Make > Save project

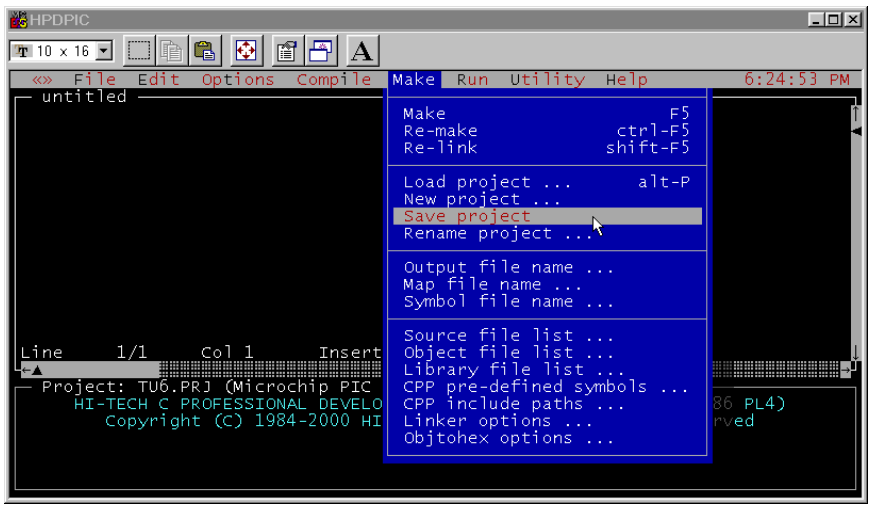


Figure 33: Saving the Project

salvocfg.h

Because the tutorial program is relatively simple, only a few configuration options need to be defined in `salvocfg.h`. By starting with an empty `salvocfg.h`, we begin with all configurations at their default values.

We're using the PICC compiler for a system running a Microchip PIC16C77. Therefore the `salvocfg.h` for this project will require

```
#define OSCOMPILER OSHT_PICC
#define OSTARGET   OSPIC16
```

in order to compile correctly. The possible values for these two preprocessor directives can be found in *Chapter 7 • Reference*, and are defined in `salvo.h`.

For three tasks and one event, we'll also need the following `#define` directives.

```
#define OSTASKS  3
#define OSEVENTS 1
```

Next, we're using messages as a means of intertask communications. Message code is disabled by default, so we enable it with:

```
#define OSENABLE_MESSAGES TRUE
```

Lastly, because we're using delays, we need to specify the size of possible delays.

```
#define OSBYTES_OF_DELAYS 1
```

This configuration option must be specified because Salvo defaults to no support for delays, which keeps RAM requirements to a minimum. Since `TaskBlink()` delays itself for 50 system ticks, a single byte is all that is required. With a byte for delays, each task could delay itself for up to 255 system ticks with a single call to `OS_Delay()`.

Note The `#defines` in `salvocfg.h` may appear in any order.

This six-line `salvocfg.h` is typical for small- to medium-sized programs of moderate complexity. The complete Salvo configuration file for this program can be found in `salvo\tut\tu6\sysa\salvocfg.h`. It is shown (with C comments removed[39]) in Listing 30.

```
#define OSBYTES_OF_DELAYS      1
#define OSCOMPILER             OSHT_PICC
#define OSENABLE_MESSAGES      TRUE
#define OSEVENTS               1
#define OSTARGET               OSPIC16
```

[39] And without the additional configuration options that match those of the associated freeware library.

```
#define OSTASKS                 3
```

Listing 30: salvocfg.h for Tutorial Program

Setting Include Paths

HPDPIC needs to locate both the project-specific configuration file `salvocfg.h` and the Salvo header file `salvo.h`. Add the two include paths to the project:

> Make > CPP include paths ...
> $(CWD)[40]
> C:\salvo\inc
> > DONE

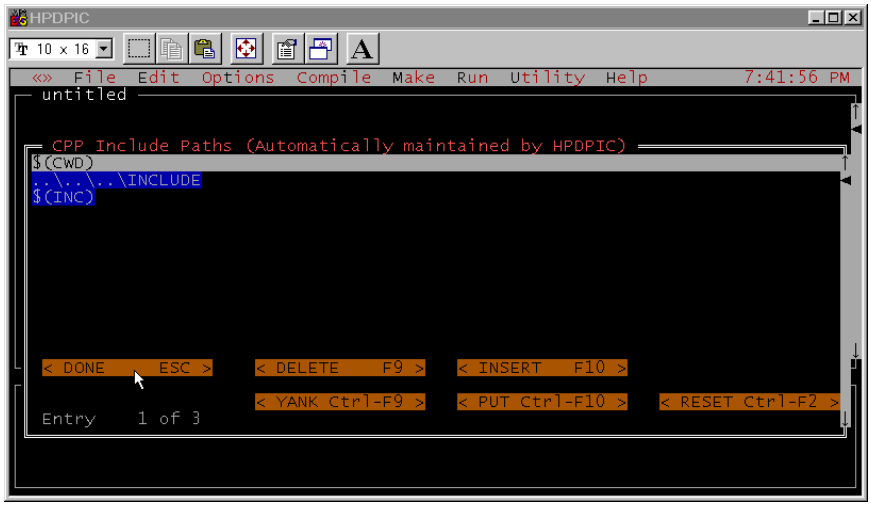


Figure 34: Setting the Include Paths

Adding Files to the Project (Freeware Libraries)

Salvo is supplied with freeware libraries for certain target processors and compilers. They are described in more detail in *Appendix C • File and Program Descriptions*. Let's compile the tutorial application using a freeware library that supports multitasking, delays and events.

Note Application Note *AN-1 Using Salvo Freeware Libraries with the HI-TECH PICC Compiler* covers this issue in greater detail.

40 Stands for Current Working Directory, i.e. where the project file is located.

If you have the full version and wish to add the source files directly as nodes in your project, please skip directly to *Adding Files to the Project (Source Code)*, below.

When using the freeware libraries for PICmicro target devices, you must always compile your `main.c` and then link to the freeware library that's appropriate for your target processor. The project already has a `main.c` – see *Figure 32: Adding the Source File main.c*, above).

If you were to compile the project at this time, the compiler would generate several link errors ("undefined symbols") – go ahead and try it:

> Make > Make

The resulting HPDPIC output is shown below:

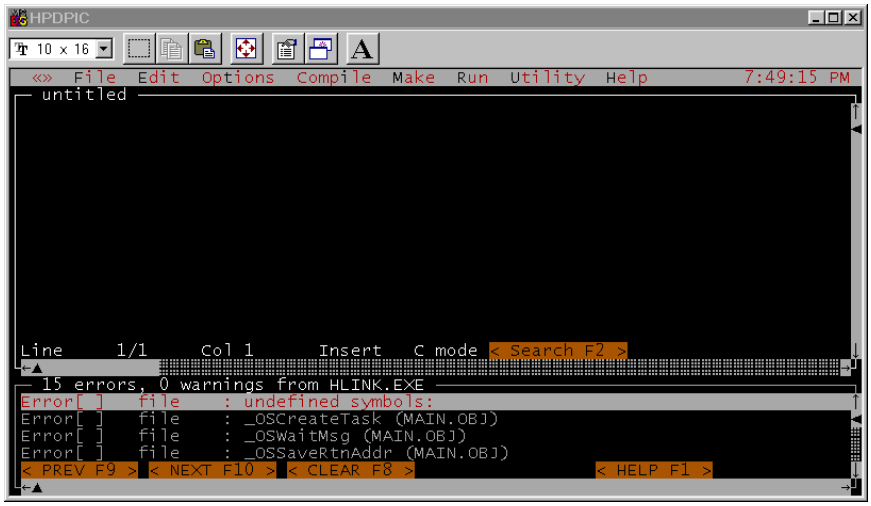


Figure 35: Link Errors due to Missing Source Files

The link errors in Figure 35 occur because the linker can't find the Salvo services called from `main.c`. Each freeware library is compiled using a specified compiler with a particular target processor in mind. The correct freeware library for this application is `sfp42Cab.lib`. It is for the Microchip PIC16C77 (among others) and was compiled using HI-TECH PICC. It can be found in `salvo\lib`.

In order to create your own application using a Salvo freeware library, you must:

- use the same `salvocfg.h` as was used to create the freeware library, and
- specify the library correctly so that your linker can find it and extract the Salvo functions from it for use in your application.

Chapter 8 • Libraries contains more information on creating a slvocfg.h for use with libraries and on how to specify the appropriate library for your application. To specify the correct library for HPDPIC to use when creating your own application, after opening your project do:

> Make > Library file list …

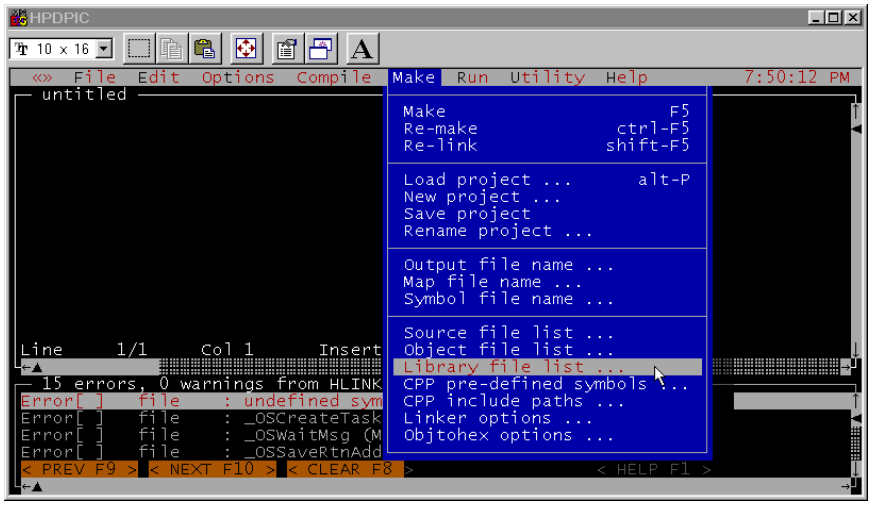


Figure 36: The Library File List

and insert the library `sfp42Cab.lib` as the first library in the Library file list …:

> INSERT (F10)
> C:\salvo\lib\sfp42Cab.lib
> Done

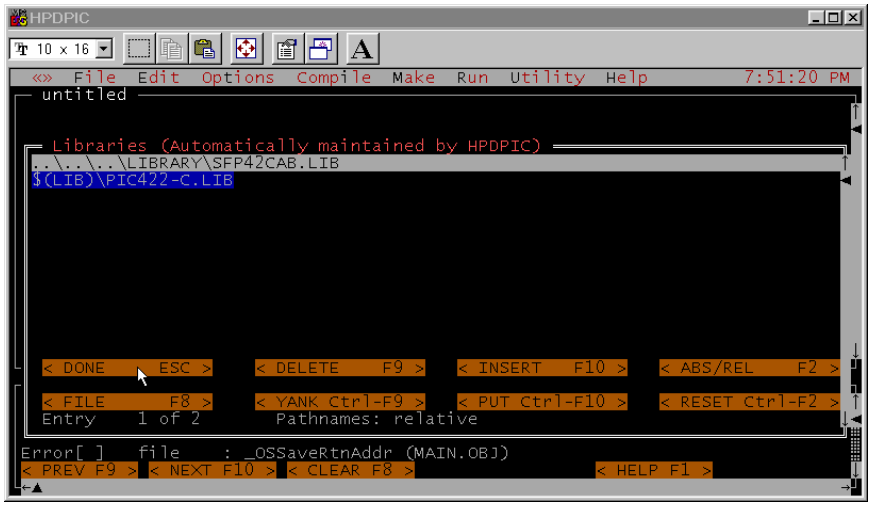


Figure 37: Freeware Library at Head of Library File List

You may need to specify an absolute or relative path for the library, depending on how you've created your project. Save the project.

> Make > Save Project

Please skip the next section and go directly to *Building the Project*, below.

Adding Files to the Project (Source Code)

If you have the full version of Salvo, you can and compile and link the Salvo files along with the main program in HPDPIC. The program in Listing 29 contains the following Salvo user services:

```
OS_Delay()
OS_Wait()
OS_Yield()
OSCreateMsg()
OSCreateTask()
OSInit()
OSSignalMsg()
OSSched()
OSTimer()
```

In *Chapter 7 • Reference* you can locate which files contain these user services. Since `main.c` includes `salvo.h` automatically, add only the files below to your project.

> Make > Source file list ...
C:\salvo\tut\tu6\main.c
C:\salvo\src\delay.c

C:\salvo\src\init.c
C:\salvo\src\inittask.c
C:\salvo\src\mem.c[41]
C:\salvo\src\msg.c
C:\salvo\src\sched.c
C:\salvo\src\timer.c
> DONE

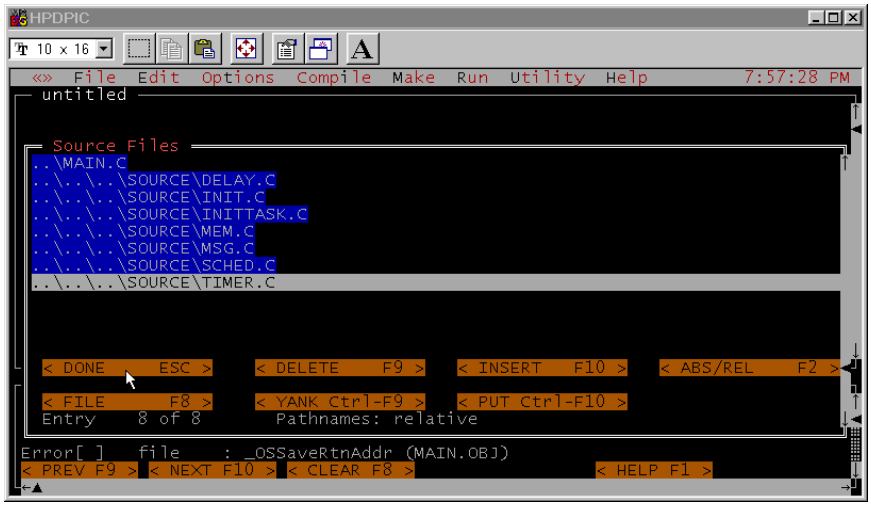


Figure 38: Adding the Obvious Source Files

If you were to compile the project at this time, the compiler would generate several link errors ("undefined symbols") – go ahead and try it:

> Make > Make

The resulting HPDPIC output is shown below:

41 Every Salvo application needs `mem.c`.

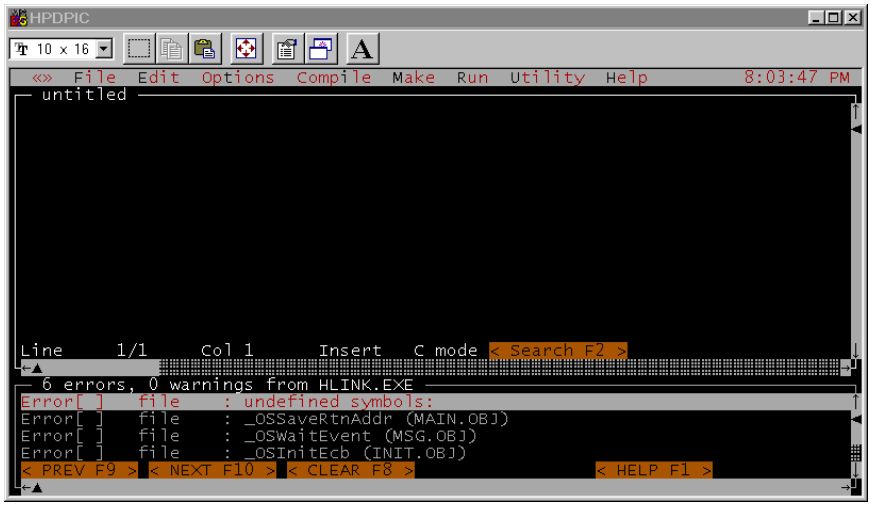


Figure 39: Link Errors due to Missing Source Files

The link errors in Figure 39 occur because the Salvo user services used in the program call Salvo functions contained in other source files. Another look in *Chapter 7 • Reference* will give the names of the other source files that also need to be linked to your application. Add them now:

> Make > Source file list ...
C:\salvo\src\event.c
C:\salvo\src\initecb.c
C:\salvo\src\inittcb.c
C:\salvo\src\qins.c
C:\salvo\src\util.c
> Done

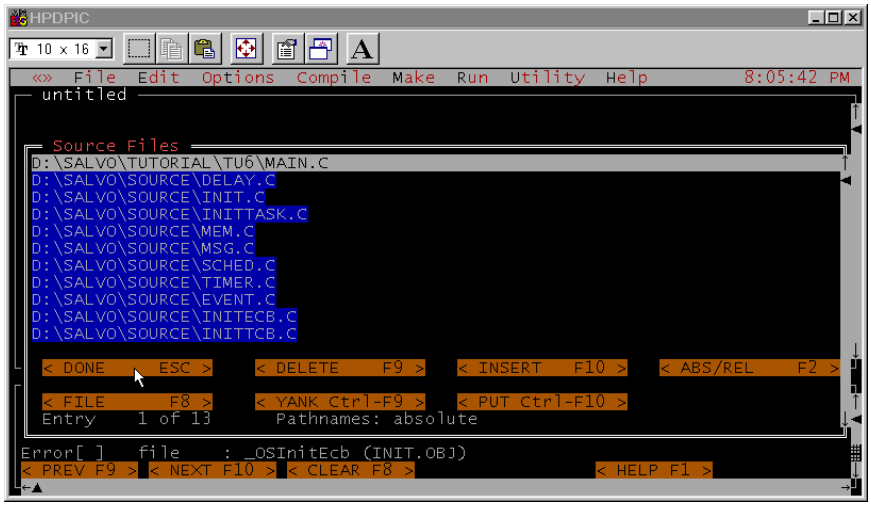


Figure 40: Complete List of Required Source Files

Save the project.

> Make > Save Project

Building the Project

Build the project.

> Make > Make

The project should compile and link successfully if you've installed Salvo and PICC correctly and followed the above instructions while paying close attention to path- and filenames:

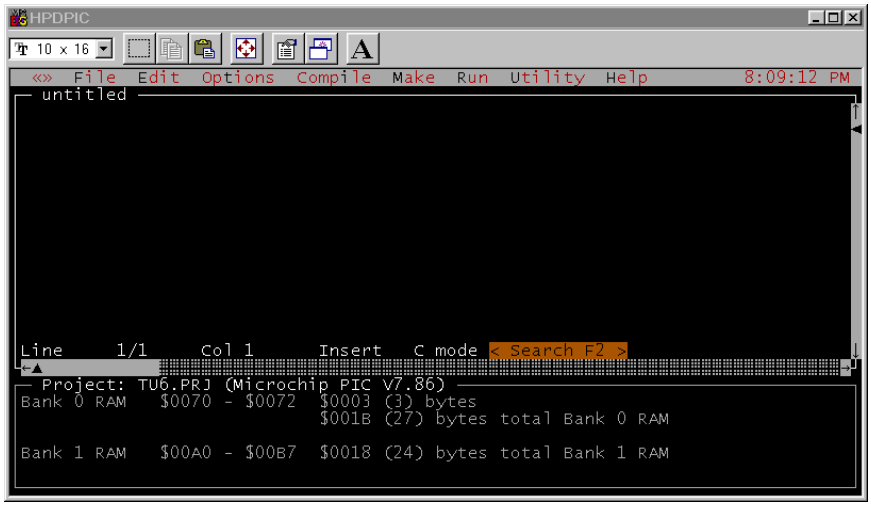


Figure 41: A Successful Compilation[42]

By clicking on the right mouse button in the lower HPDPIC window, you can expand it to full-screen and look at the memory requirements of the program. Totals for the amount of ROM and RAM are displayed:

42 Results for `tu6.prj`.

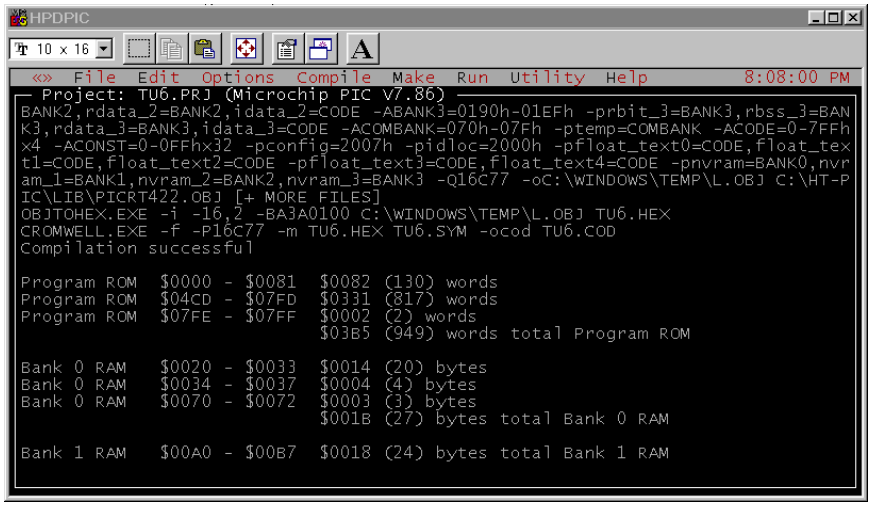


Figure 42: Display of Memory Usage

Examining Memory Usage

The PICC compiler generates a map file, `salvo\tut\tu6\sysa\tu6.map`. By examining it you can find:

- where Salvo's functions are located in ROM,
- where Salvo's data structures are located in RAM,
- the size of Salvo function,
- the call tree in your application,
- and many other things.

This information can be useful when you are debugging your application, or if you just want to find out how Salvo works. When using some of Salvo's more advanced configuration options you may want to refer back to the map file to see the effects on ROM and RAM of the new configuration(s) you have selected.

Running the Tutorial Program – PIC16C77 and Similar PICmicros

The target-specific tutorial program projects `tu1*-tu6*.pjt\sysa` use the Microchip PIC16C77 PICmicro. They will also run on the following members of the PIC16 family without changes:

PIC16C66
PIC16C67
PIC16C745
PIC16C76
PIC16C765
PIC16C77

PIC16F76
PIC16F77
PIC16F876
PIC16F877

If you have a device programmer, you can program any one of the PICmicro devices above with the file `salvo\tut\tu6\sysa\tu6.hex`.[43] Install the chip onto a PICDEM-2 demonstration board,[44] provide a 4MHz crystal oscillator and power to the board, and the program should begin running. The LED on `PORT`:0 will blink at 1Hz, and the other 7 LEDs will reflect the instantaneous value of the upper 7 bits of the free-running 16-bit counter.

Running the Tutorial Program – Other PIC16 PICmicros

You can also run these tutorials on other members of the PIC16 family. You must recompile the project after changing the library to the library appropriate for the processor you have. For instance, to compile and run `tu4.pjt` on a PIC16F84, you will need to replace the library `sfp42Cab.lib` with `sfp40Aab.lib` in the project's nodes. See *Chapter 8 • Libraries* and *AN-1 Using Salvo Freeware Libraries with the HI-TECH PICC Compiler* for more information.

Downloading and Debugging with MPLAB

If you are using HPDPIC and have Microchip's MPLAB installed on your computer, you can configure MPLAB to use the output of the compiler for downloading and symbolic debugging.

Note MPLAB may be unable to trace through Salvo source files (see Figure 46, below) if you've used the relative pathname option in HPDPIC.[45] Use absolute pathnames instead – see Figure 40, above.

First, create an MPLAB project in the same directory as your PICC project:

> Project > New Project

[43] `tu6free.hex` if you've compiled with freeware libraries.

[44] All that is needed for the tutorial project is a set of LEDs connected to the PICmicro's `PORTB`.

[45] See Figure 38.

select c:\salvo\tut\tu6\sysa
name the project tu6.pjt
> OK

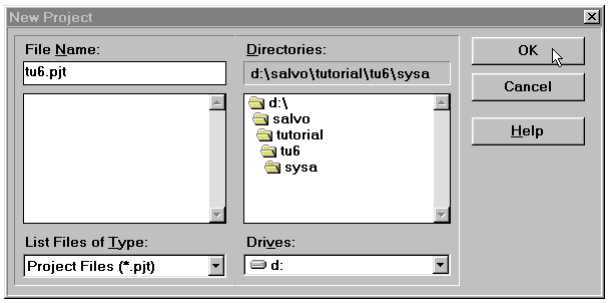


Figure 43: Creating a New Project in MPLAB

Accept the defaults for the new project:

> OK

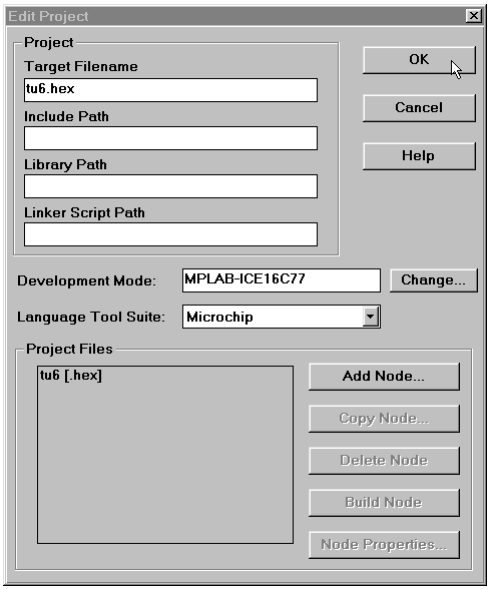


Figure 44: Accepting Project Defaults in MPLAB

Save the project:

Project > Save Project

Now you can download the hex file directly to a PICMASTER in-circuit emulator:

> File > Import > Download To Memory
tu6.hex
> OK

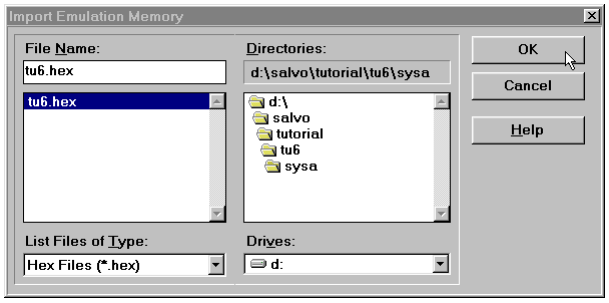


Figure 45: Downloading the Program

After each download, the processor should be reset. Then let it run:

> Debug > Run > Reset
> Debug > Run > Run

You can stop and start the program at any time with MPLAB, and it will display where you are within your application or within the Salvo source code.[46] You can also access Salvo's variables symbolically through watch windows, etc.

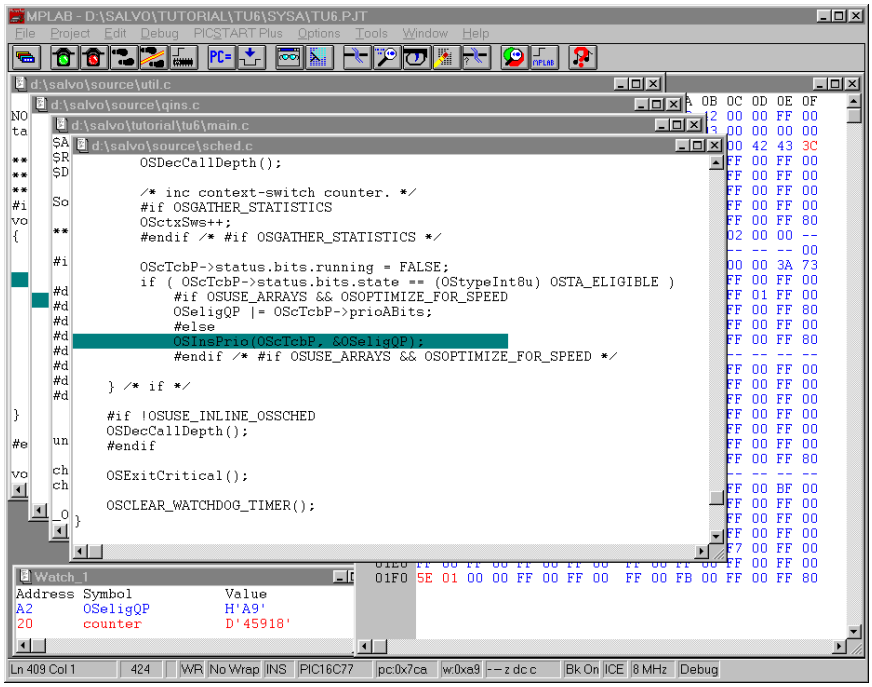


Figure 46: Symbolic Debugging via MPLAB

46 In order to trace execution in the project's source files, a source file must be open and must be the active (front) window inside MPLAB. Tracing through Salvo's source files is only possible with a source code build. Applications built with freeware or standard libraries will only trace through user source code.

Chapter 5 • Configuration

Introduction

The Salvo source code contains configuration options that you can use to tailor its linkable object code to the specific needs of your application. These options are used to identify the compiler you're using and the processor you're compiling for, to configure Salvo for the number of tasks and events your application will require, and to enable or disable support for certain services. By selecting various configuration options you can fine-tune Salvo's abilities and performance to best match your application.

Note All configuration options are in the form of C preprocessor `#define` statements. They are therefore *compile-time* options. This means that they will not take effect until / unless you recompile each Salvo source code file that is affected by the configuration option.

Overview

This section describes the Salvo configuration options. Each description includes information on:

- the name of the configuration option,
- the purpose of the configuration option,
- the allowed values for the configuration option,
- the default value for the configuration option,
- the compile-time action that results from the configuration option,
- which Salvo C source or include files contain the configuration option,
- related configuration options,
- which user services are enabled by the configuration option,
- how it affects memory requirements[47] and
- notes particular to the configuration option.

[47] ROM requirements are described as small (e.g. a few lines of code in a single function) to considerable (e.g. a few lines of code in nearly every function).

You can fine-tune Salvo's capabilities, performance and size by choosing configuration options appropriate to your application.

Note All configuration options are contained in the user file `salvocfg.h`, and should not be placed in any other file(s). `salvocfg.h` should be located in the same directory as your application's source files. See *Chapter 4 • Tutorial* for more information on `salvocfg.h`.

Caution Whenever a configuration option is changed in `salvocfg.h`, you must recompile all of the Salvo files in your application. Failing to do so may result in unpredictable behavior or erroneous results.

Organization

The configuration options are loosely organized as outlined below, by category.

Compiler in use:	`OSCOMPILER`
Target processor:	`OSTARGET`
Tasks and events:	`OSBIG_MESSAGE_POINTERS`, `OSBIG_SEMAPHORES`, `OSENABLE_BINARY_SEMAPHORES`, `OSENABLE_EVENT_READING`, `OSENABLE_EVENT_TRYING`, `OSENABLE_FAST_SIGNALING`, `OSENABLE_IDLE_COUNTER`, `OSENABLE_IDLING_HOOK`, `OSENABLE_MESSAGES`, `OSENABLE_MESSAGE_QUEUES`, `OSENABLE_SEMAPHORES`, `OSEVENTS`, `OSTASKS`, `OSMESSAGE_QUEUES`
Size-specific:	`OSBYTES_OF_COUNTS`, `OSBYTES_OF_DELAYS`, `OSBYTES_OF_EVENT_FLAGS`, `OSBYTES_OF_TICKS`
Time and ticks:	`OSENABLE_TIMEOUTS`, `OSTIMER_PRESCALAR`
Optimizations:	`OSCLEAR_GLOBALS`, `OSOPTIMIZE_FOR_SPEED`, `OSSPEEDUP_QUEUEING`, `OSUSE_OSINSELIGQ_MACRO`

Monitor and debugging:	OSCLEAR_UNUSED_POINTERS, OSENABLE_STACK_CHECKING, OSLOGGING, OSLOG_MESSAGES, OSRPT_HIDE_INVALID_POINTERS, OSRPT_SHOW_ONLY_ACTIVE, OSRPT_SHOW_TOTAL_DELAY
Error checking:	OSDISABLE_ERROR_CHECKING, OSUSE_EVENT_TYPES
Statistics:	OSGATHER_STATISTICS
Memory allocation and RAM banking:	OSLOC_ALL, OSLOC_COUNT, OSLOC_CTCB, OSLOC_DEPTH, OSLOC_ECB, OSLOC_ERR, OSLOC_LOGMSG, OSLOC_MQCB, OSLOC_MSGQ, OSLOC_PS, OSLOC_SIGQ, OSLOC_TCB, OSLOC_TICK, OSMPLAB_C18_LOC_ALL_NEAR, OSUSE_CHAR_SIZED_BITFIELDS, OSUSE_MEMSET
Interrupts:	OSCALL_OSCREATEEVENT, OSCALL_OSMSGQCOUNT, OSCALL_OSMSGQEMPTY, OSCALL_OSRETURNEVENT, OSCALL_OSSIGNALEVENT, OSCALL_OSSTARTTASK, OSINTERRUPT_LEVEL, OSPRESERVE_INTERRUPT_MASK, OSTIMER_PRESCALAR
Hardware issues:	OSCLEAR_WATCHDOG_TIMER(), OSPIC16_GIE_BUG, OSPIC18_INTERRUPT_MASK
Porting:	OSCTXSW_METHOD, OSRTNADDR_OFFSET
Stack depth usage:	OSUSE_INLINE_OSSCHED, OSUSE_INLINE_OSTIMER
Code compression:	OSCOMBINE_EVENT_SERVICES
Linking to libraries:	OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_TYPE, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Hooks to user code:	OSENABLE_IDLING_HOOK, OSENABLE_INTERRUPT_HOOKS, OSENABLE_SCHEDULER_HOOK
Scheduler behavior:	OSDISABLE_FAST_SCHEDULING
Extensions:	OSENABLE_TCBEXT0|1|2|3|4|5, OSTYPE_TCBEXT0|1|2|3|4|5

Table 2: Configuration Options by Category

Choosing the Right Options for your Application

You must select a compiler and a target when configuring Salvo for your application. Depending on how many Salvo services you wish to use in your application, you will also need to select and/or configure other options. Consult the table below for further information:

Multitasking:	OSTASKS
Using events:	OSENABLE_BINARY_SEMAPHORES, OSENABLE_EVENT_FLAGS, OSENABLE_FAST_SIGNALING, OSENABLE_MESSAGES, OSENABLE_MESSAGE_QUEUES, OSENABLE_SEMAPHORES, OSEVENTS
Using multiple event types:	OSCOMBINE_EVENT_SERVICES
Keeping unused code out of your application:	OSENABLE_EVENT_READING, OSENABLE_EVENT_TRYING
Delaying tasks:	OSBYTES_OF_DELAYS
Waiting on events with a timeout:	OSBYTES_OF_DELAYS
Setting the size of event flags:	OSBYTES_OF_EVENT_FLAGS
Keeping track of elapsed time:	OSBYTES_OF_TICKS
Counting the number of context switches:	OSBYTES_OF_COUNTS, OSGATHER_STATISTICS
Using 16-bit semaphores:	OSBIG_SEMAPHORES
Using ROM and RAM pointers:	OSBIG_MESSAGE_POINTERS
Having an idle function:	OSENABLE_IDLING_HOOK, OSENABLE_IDLE_COUNTER
Checking call ... return stack depth:	OSENABLE_STACK_CHECKING, OSGATHER_STATISTICS
Collecting statistics:	OSGATHER_STATISTICS
Logging descriptive error, warning and status messages:	OSLOGGING, OSLOG_MESSAGES
Optimizing your application:	OSCLEAR_GLOBALS, OSOPTIMIZE_FOR_SPEED, OSSPEEDUP_QUEUEING

Making the most of limited resources:	OSTIMER_PRESCALAR
Avoiding event-type mismatches:	OSUSE_EVENT_TYPES
Learning how Salvo works:	OSCLEAR_UNUSED_POINTERS, OSRPT_HIDE_INVALID_POINTERS, OSRPT_SHOW_ONLY_ACTIVE, OSRPT_SHOW_TOTAL_DELAY
Porting to other compilers and / or target processors:	OSCTXSW_METHOD, OSRTNADDR_OFFSET, OSUSE_MEMSET
Minimizing Salvo's call…return stack usage:	OSUSE_INLINE_OSSCHED, OSUSE_INLINE_OSTIMER
Calling Salvo services from the background and the foreground:	OSCALL_OSCREATEEVENT, OSCALL_OSMSGQCOUNT, OSCALL_OSMSGQEMPTY, OSCALL_OSRETURNEVENT, OSCALL_OSSIGNALEVENT, OSCALL_OSSTARTTASK
Locating Salvo's variables in memory:	OSLOC_ALL, OSLOC_COUNT, OSLOC_CTCB, OSLOC_DEPTH, OSLOC_ECB, OSLOC_ERR, OSLOC_LOGMSG, OSLOC_MQCB, OSLOC_MSGQ, OSLOC_PS, OSLOC_SIGQ, OSLOC_TCB, OSLOC_TICK, OSMPLAB_C18_LOC_ALL_NEAR
Building an application with the freeware libraries:	OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_TYPE, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Running multiple tasks at same priority (round-robin):	OSDISABLE_FAST_SCHEDULING
Minimizing memory usage:	OSUSE_CHAR_SIZED_BITFIELDS
Extending task-specific functionality:	OSENABLE_TCBEXT0|1|2|3|4|5, OSTYPE_TCBEXT0|1|2|3|4|5

Table 3: Configuration Options by Desired Feature

Creating and Editing the Configuration File

You can create and edit your project's configuration file `salvocfg.h` with any text editor. A sample file can be found in `salvo\inc\user`.

`salvocfg.h` should be located in the same directory as your application's source files.

Predefined Configuration Constants

Predefined symbols are listed with their values below.

FALSE	0
TRUE	1
OSLOG_NONE, OSLOG_ERRORS, OSLOG_WARNINGS, OSLOG_ALL	see OSLOG_MESSAGES
OSUNDEF, OSNONE	0
OSPIC12, OSPIC16, OSPIC17, OSPIC18, OSIX86, OSI8051, OSM68HC11, OSMSP430, OSVAV8	see OSTARGET
OSAQ_430, OSGCC, OSHT_8051C, OSHT_PICC, OSHT_V8C, OSIMAGECRAFT, OSMW_CW, OSMIX_PC, OSIAR_ICC, OSMPLAB_C18, OSKEIL_C51	see OSCOMPILER
OSFROM_-BACKGROUND, OSFROM_-FOREGROUND, OSFROM_ANYWHERE	see OSCALL_XYZ
OSRTNADDR_IS_-PARAM, OSRTNADDR_IS_VAR, OSVIA_OSCTXSW, OSVIA_OSDISPATCH	see OSCTXSW_METHOD

OSALL_BITS, OSANY_BITS, OSEXACT_BITS	see OS_WaitEFlag()
OSA, OSB, …, OSZ	see OSLIBRARY_CONFIG, OSLIBRARY_TYPE, and OSLIBRARY_VARIANT

Table 4: Predefined Symbols

Configuration Options for all Distributions

The configuration options described in this section can be used with:

- Salvo Lite
- Salvo LE
- Salvo Pro
- Salvo Developer

and are listed in alphabetical order.

These configuration options affect the Salvo header (*.h) files, as well as mem.c.

OSCOMPILER: Identify Compiler in Use

Name:	`OSCOMPILER`
Purpose:	To identify the compiler you're using to generate your Salvo application.
Allowed Values:	`OSAQ_430`: Archelon Quadravox AQ430 `OSGCC`: GNU C Compiler (gcc) `OSHT_8051C`: HI-TECH 8051C `OSHT_PICC`: HI-TECH PICC and PICC-18 `OSHT_V8C`: HI-TECH V8C `OSIAR_ICC`: IAR C compilers `OSIMAGECRAFT`: ImageCraft compilers `OSKEIL_C51`: Keil C51 `OSMIX_PC`: Mix Power C `OSMPLAB_C18`: Microchip MPLAB-C18 `OSMW_CW`: Metrowerks CodeWarrior
Default Value:	`OSUNDEF`, or automatically defined for certain compilers.
Action:	Configures Salvo source code for use with the selected compiler.
Contained in:	`salvo.h`, `multcall.h`, `init.c`, `timer.c`
Related:	`OSTARGET`
Enables:	–
Memory Required:	n/a

Notes

This configuration option is used within the Salvo source code primarily to implement non-ANSI C directives like in-line assembly instructions and `#pragma` directives.

Salvo automatically detects the presence of the compilers listed below, and sets `OSCOMPILER` accordingly.[48]

- Archelon Quadravox AQ430
- GNU C Compiler (gcc)
- HI-TECH 8051C
- HI-TECH PICC, PICC-Lite and PICC-18
- HI-TECH V8C
- IAR MSP430 C
- IAR PIC18 C
- Keil C51
- Metrowerks CodeWarrior
- Microchip MPLAB-C18

[48] `OSCOMPILER` can be overridden by setting it in `salvocfg.h`.

The compiler you're using must be identified via this configuration option, either automatically or via an entry in `salvocfg.h`.

If you are working with an as-yet-unsupported compiler, use `OSUNDEF` and refer to *Chapter 10 • Porting* for further instructions.

OSEVENTS: Set Maximum Number of Events

Name:	`OSEVENTS`
Purpose:	To allocate memory at compile time for event control blocks (ecbs), and to set an upper limit on the number of supported events.
Allowed Values:	`0` or greater.
Default Value:	`0`
Action:	Configures Salvo source code to support the desired number of events.
Contained in:	`salvo.h`, `chk.c`, `init.c`, `mem.c`, `rpt.c`, `util.c`
Related:	`OSENABLE_BINARY_SEMAPHORES`, `OSENABLE_EVENTS`, `OSENABLE_MESSAGES`, `OSENABLE_MESSAGE_QUEUES`, `OSENABLE_SEMAPHORES`, `OSEVENT_FLAGS`, `OSTASKS`, `OSMESSAGE_QUEUES`
Enables:	`OSCreateBinSem()`, `OSCreateMsg()`, `OSCreateMsgQ()`, `OSCreateSem()`, `OSSignalBinSem()`, `OSSignalMsg()`, `OSSignalMsgQ()`, `OSSignalSem()`, `OS_WaitBinSem()`, `OS_WaitMsg()`, `OS_WaitMsgQ()`, `OS_WaitSem()`
Memory Required:	When non-zero, requires a configuration-dependent amount of RAM for each ecb.

Notes

Events (event flags, all semaphores, messages and message queues) are numbered from `1` to `OSEVENTS`.

Since event memory is allocated at compile time, the ecb memory will be used whether or not the event is actually created via `OSCreateBinSem/Msg/MsgQ/Sem()`.

On a typical 8-bit processor, the amount of memory required by each event is 2-4 bytes[49] depending on which configuration options are enabled.

[49] For the purposes of these size estimates, pointers to ROM memory are assumed to be 16 bits, and pointers to RAM memory are assumed to be 8 bits. This is the situation for the PIC16 and PIC17 family of processors.

OSEVENT_FLAGS: Set Maximum Number of Event Flags

Name:	OSEVENT_FLAGS
Purpose:	To allocate memory at compile time for event flag control blocks (efcbs), and to set an upper limit on the number of supported event flags.
Allowed Values:	1 or greater.
Default Value:	1 if OSENABLE_EVENT_FLAGS is TRUE, 0 otherwise
Action:	Configures Salvo source code to support the desired number of event flags.
Contained in:	salvo.h, mem.c
Related:	OSENABLE_EVENT_FLAGS, OSLOC_EFCB,
Enables:	-
Memory Required:	When non-zero, requires a configuration-dependent amount of RAM for each efcb.

Notes

This configuration parameter allocates RAM for event flag control blocks. Event flags require no other additional memory.

Event flags are numbered from 1 to OSEVENT_FLAGS.

Since event flag memory is allocated at compile time, the efcb memory will be used whether or not the event flag is actually created via OSCreateEFlag().

On a typical 8-bit processor, the amount of memory required by each event flag control block is represented by OSBYTES_OF_EVENT_FLAGS.

OSLIBRARY_CONFIG: Specify Precompiled Library Configuration

Name:	OSLIBRARY_CONFIG
Purpose:	To guarantee that an application's source files are compiled using the same salvocfg.h as was used to create the specified precompiled library.
Allowed Values:	certain ones in the range OSA through OSZ
Default Value:	not defined
Action:	Sets the configuration options inside salvolib.h to match those used to generate the library specified.
Contained in:	salvo.h, salvolib.h
Related:	OSLIBRARY_TYPE, OSLIBRARY_GLOBALS, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Enables:	–
Memory Required:	n/a

Notes

OSLIBRARY_CONFIG is used in conjunction with OSLIBRARY_GLOBALS, OSLIBRARY_TYPE, OSLIBRARY_VARIANT and OSUSE_LIBRARY to properly specify the precompiled Salvo library you're linking to your project.

Library configurations might refer to, for example, whether the library is configured to support delays and/or events.

Please see the documentation accompanying the library of interest for the allowed values of OSLIBRARY_CONFIG.

See Also

The examples under OSUSE_LIBRARY

OSLIBRARY_GLOBALS: Specify Memory Type for Global Salvo Objects in Precompiled Library

Name:	OSLIBRARY_GLOBALS
Purpose:	To guarantee that an application's source files are compiled using the same salvocfg.h as was used to create the specified precompiled library.
Allowed Values:	certain ones in the range OSA through OSZ
Default Value:	not defined
Action:	Sets the configuration options inside salvolib.h to match those used to generate the library specified.
Contained in:	salvo.h, salvolib.h
Related:	OSLIBRARY_TYPE, OSLIBRARY_CONFIG, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Enables:	–
Memory Required:	n/a

Notes

OSLIBRARY_GLOBALS is used in conjunction with OSLIBRARY_CONFIG, OSLIBRARY_TYPE, OSLIBRARY_VARIANT and OSUSE_LIBRARY to properly specify the precompiled Salvo library you're linking to your project.

Library configurations might refer to, for example, whether the library is configured to support delays and/or events.

Please see the documentation accompanying the library of interest for the allowed values of OSLIBRARY_GLOBALS.

See Also

The examples under OSUSE_LIBRARY

OSLIBRARY_TYPE: Specify Precompiled Library Type

Name:	OSLIBRARY_TYPE
Purpose:	To guarantee that an application's source files are compiled using the same salvocfg.h as was used to create the specified precompiled library.
Allowed Values:	certain ones in the range OSA through OSZ
Default Value:	not defined
Action:	Sets the configuration options inside salvolib.h to match those used to generate the library specified.
Contained in:	salvo.h, salvolib.h
Related:	OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Enables:	–
Memory Required:	n/a

Notes

OSLIBRARY_TYPE is used in conjunction with OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_VARIANT and OSUSE_LIBRARY to properly specify the precompiled Salvo library you're linking to your project.

Library types might refer to, for example, whether the library is is a freeware library or a standard library.

Please see the documentation accompanying the library of interest for the allowed values of OSLIBRARY_TYPE.

See Also

The examples under OSUSE_LIBRARY

OSLIBRARY_VARIANT: Specify Precompiled Library Variant

Name:	OSLIBRARY_VARIANT
Purpose:	To guarantee that an application's source files are compiled using the same salvocfg.h as was used to create the specified precompiled library.
Allowed Values:	certain ones in the range OSA through OSZ, and OSNONE
Default Value:	not defined
Action:	Sets the configuration options inside salvolib.h to match those used to generate the library specified.
Contained in:	salvo.h, salvolib.h
Related:	OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_TYPE, OSUSE_LIBRARY
Enables:	–
Memory Required:	n/a

Notes

OSLIBRARY_VARIANT must be used in conjunction with OSLIBRARY_GLOBALS, OSLIBRARY_CONFIG, OSLIBRARY_TYPE and OSUSE_LIBRARY to properly specify the precompiled Salvo library you're linking to your project.

Library variants might refer to, for example, whether the library supports signaling events from within ISRs.

Not all libraries have variants. If a variant does not exist, set OSLIBRARY_VARIANT to OSNONE.

Please see the documentation accompanying the library of interest for the allowed values of OSLIBRARY_VARIANT.

See Also

The examples under OSUSE_LIBRARY

OSMESSAGE_QUEUES: Set Maximum Number of Message Queues

Name:	`OSMESSAGE_QUEUES`
Purpose:	To allocate memory at compile time for message queue control blocks (mqcbs), and to set an upper limit on the number of supported message queues.
Allowed Values:	`1` or greater.
Default Value:	`1` if `OSENABLE_MESSAGE_QUEUES` is `TRUE`, `0` otherwise
Action:	Configures Salvo source code to support the desired number of message queues.
Contained in:	`salvo.h`, `chk.c`, `mem.c`
Related:	`OSENABLE_MESSAGE_QUEUES`, `OSLOC_MQCB`, `OSLOC_MSGQ`
Enables:	`OSCreateMsgQ()`, `OSSignalMsgQ()`, `OS_WaitMsgQ()`
Memory Required:	When non-zero, requires a configuration-dependent amount of RAM for each mqcb.

Notes

This configuration parameter only allocates RAM for message queue control blocks. It does not allocate RAM for the message queues themselves – you must do that explicitly.

Message queues are numbered from `1` to `OSMESSAGE_QUEUES`.

Since message queue memory is allocated at compile time, the mqcb memory will be used whether or not the message queue is actually created via `OSCreateMsgQ()`.

On a typical 8-bit processor, the amount of memory required by each message queue control block is 6 bytes.

OSTARGET: Identify Target Processor

Name:	`OSTARGET`
Purpose:	To identify the processor you're using in your Salvo application.
Allowed Values:	`OSM68HC11`: Motorola M68HC11 microcontroller `OSMSP430`: TI's MSP430 ultra-low-power microcontroller `OSPIC12`: Microchip's 12-bit family (e.g. PIC16C57) of PICmicro MCUs `OSPIC16`: Microchip's 14-bit family (e.g. PIC16C67) of PICmicro MCUs `OSPIC17`: Microchip's 16-bit family (e.g. PIC17C756A) of PICmicro MCUs `OSPIC18`: Microchip family (e.g. PIC18C452) of PICmicro MCUs `OSIX86`: Intel x86 family (e.g. 80386, Pentium, etc.) `OSI8051`: Intel 8051 family and derivatives `OSVAV8`: VAutomation V8-µRISC™
Default Value:	`NONE`
Action:	Configures Salvo source code for the target processor.
Contained in:	`portXyz.h`, `salvo.h`
Related:	`OSCOMPILER`
Enables:	–
Memory Required:	n/a

Notes

This configuration option is used within the Salvo source code primarily to implement non-ANSI C directives like in-line assembly instructions and `#pragma` directives.

You must identify the processor you're using via this configuration option. However, some compilers (e.g. HI-TECH PICC) automatically override your settings and define `OSTARGET` based on the command-line arguments passed to the compiler to identify the processor.

If you are working with an as-yet-unsupported compiler, choose `OSUNDEF`. See *Chapter 10 • Porting* for more information.

OSTASKS: Set Maximum Number of Tasks

Name:	`OSTASKS`
Purpose:	To allocate memory at compile time for task control blocks (tcbs), and to set an upper limit on the number of supported tasks.
Allowed Values:	`1` or greater.
Default Value:	`0`
Action:	Configures Salvo source code to support the desired number of tasks.
Contained in:	`salvo.h`, `chk.c`, `init.c`, `mem.c`, `rpt.c`, `util.c`
Related:	`OSEVENTS`
Enables:	`OSInit()`, `OSCreateTask()`, `OSSched()`, `OSStartTask()`
Memory Required:	When non-zero, requires a configuration-dependent amount of RAM for each tcb, and RAM for two tcb pointers.

Notes

Tasks are numbered from `1` to `OSTASKS`.

Since task memory is allocated and fixed at compile time, the tcb memory will be used whether or not the task is actually created via `OSCreateTask()`.

The amount of memory required by each task is dependent on several configuration options, and will range from a minimum of 4 to a maximum 12 bytes per task.[50]

[50] For the purposes of these size estimates, pointers to ROM memory are assumed to be 16 bits, and pointers to RAM memory are assumed to be 8 bits. This is the situation for the PIC16 and PIC17 family of processors.

OSUSE_LIBRARY: Use Precompiled Library

Name:	OSUSE_LIBRARY
Purpose:	To simplify linking to a precompiled Salvo library.
Allowed Values:	FALSE: you are not linking to a precompiled Salvo library. TRUE: you are linking to a precompiled Salvo library.
Default Value:	FALSE
Action:	If TRUE, the proper configuration options for the specified library will be used to build the application.
Contained in:	salvo.h, salvolib.h
Related:	OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_TYPE, OSLIBRARY_VARIANT
Enables:	–
Memory Required:	n/a

Notes

Salvo's configuration options are *compile-time options*. When linking to a precompiled library of Salvo services, the settings for your own application *must match* those originally used when the library was generated. OSUSE_LIBRARY takes the guesswork out of creating a salvocfg.h header file for each library you use.

Warning Failure to have matching configuration options may lead to compile- and link-time errors that can be difficult to interpret. Because of the large number of configuration options and their interrelationships, you *must* use OSUSE_LIBRARY when linking to precompiled Salvo libraries.

In order to simplify linking to precompiled Salvo libraries (freeware or standard), instead of creating a salvocfg.h file for your application, just set OSUSE_LIBRARY to TRUE, and specify the appropriate library type, configuration and variant codes via additional configuration options. Then link to the appropriate library when building / making the project.

Configuration options used to create precompiled Salvo libraries differ from library to library. Please see the documentation that accompanies each library for its configuration options.

For example, to include a precompiled freeware library for use with the HI-TECH PICC compiler and PIC16C77 PICmicro MCU

(sfp42Cab.lib, configuration code f for *f*reeware library, p42C for *PIC16C77* and equivalent)[51] that supports full multitasking and with delays and events (configuration code a for *a*ll) and event signaling services to be called strictly from the background (configuration code b for *b*ackground), your salvocfg.h should contain just these four entries:[52]

```
#define OSUSE_LIBRARY      TRUE
#define OSLIBRARY_TYPE     OSF
#define OSLIBRARY_CONFIG   OSA
#define OSLIBRARY_VARIANT  OSB
```

OSUSE_LIBRARY does not link any library to your application. Rather, it configures the salvolib.h header file and any source (*.c) files in your project to be consistent with the configuration options used to create the library. It does this by including salvolib.h in each of the files mentioned above.

You must specify the library you want to link to separately, either as a compiler / linker directive or as project setting (e.g. a library node in an MPLAB project),

If you are using a precompiled library that enables you to specify certain configuration options that differ[53] from / override the library's default(s), you may also add them to your salvocfg.h, like this:

```
#define OSUSE_LIBRARY      TRUE
#define OSLIBRARY_TYPE     OSF
#define OSLIBRARY_CONFIG   OSA
#define OSLIBRARY_VARIANT  OSB

#define OSEVENT_FLAGS      0
#define OSEVENTS           1
#define OSMESSAGE_QUEUES   0
#define OSTASKS            3
```

You should not use OSUSE_LIBRARY in conjunction with libraries you generate – only with precompiled libraries supplied as part of the standard Salvo distribution.

---

51 This Salvo (s) freeware (f) library is compiled with PICC (p) for the PIC16C77 PICmicro (42C) with all functions enabled (a) and event signaling from the background (b) only.

52 For most precompiled libraries it's unnecessary to specify the compiler or target processor type in your salvocfg.h.

53 With freeware libraries, those configuration parameters can range from 0 to the maximum supported in the library. With standard libraries, the maximum value is limited only by available memory.

See *AN-1 Using Salvo Freeware Libraries with the HI-TECH PICC Compiler* for more information.

Configuration Options for Source Code Distributions

The configuration options described in this section can only be used with:

- Salvo Pro
- Salvo Developer

and are listed in alphabetical order.

These configuration options affect the Salvo header (`*.h`) and source (`*.c`) files.

OSBIG_MESSAGE_POINTERS: Allow Message Pointers to RAM and ROM

Name:	OSBIG_MESSAGE_POINTERS
Purpose:	Enable message pointers to access ROM as well as RAM. Compiler-dependent.
Allowed Values:	FALSE: Message pointers defined as void pointers. TRUE: Message pointers have a compiler-dependent definition in portxyz.h.
Default Value:	FALSE
Action:	Redefines the defined type OStypeMsg.
Contained in:	salvo.h, portpicc.h
Related:	OSCOMPILER
Enables:	n/a
Memory Required:	When TRUE, requires additional RAM in each ecb.

Notes

Salvo's message pointers (of type OStypeMsgP), used by messages and message queues, are normally defined as void pointers, i.e. `void *`. A void pointer can usually point to anywhere in RAM or ROM. This is useful, for instance, if some of your message pointers point to constant strings in ROM as well as static variables (in RAM).

Some supported compilers require an alternate definition for message pointers in order to point to ROM and RAM. Setting OSBIG_MESSAGE_POINTERS to TRUE causes message pointers to be defined in an alternate manner in order to point to both RAM and ROM.

For the HI-TECH PICC compiler, when TRUE, message pointers will be defined as `const *` instead of `void *`.

If you have defined OSBIG_MESSAGE_POINTERS to be TRUE, you may want to select OSBIG_SEMAPHORES to be TRUE also. Either configuration option, when TRUE, will enlarge the size of ecbs, typically by one byte.

OSBIG_SEMAPHORES: Use 16-bit Semaphores

Name:	`OSBIG_SEMAPHORES`
Purpose:	To select 8- or 16-bit counting semaphores.
Allowed Values:	`FALSE`: Counting semaphores range from 0 to 255. `TRUE`: Counting semaphores range from 0 to 32,767.
Default Value:	`FALSE`
Action:	Changes the defined type `OStypeSem` from 8- to 16-bit unsigned integer.
Contained in:	`salvo.h`, `sem.c`
Related:	–
Enables:	n/a
Memory Required:	When `TRUE`, requires an additional byte of RAM for each ecb.

Notes

This configuration option can be used to minimize the size of ecbs. Make `OSBIG_SEMAPHORES TRUE` only if your application requires 16-bit counting semaphores.

If you have selected `OSBIG_MESSAGE_POINTERS` to be `TRUE`, setting `OSBIG_SEMAPHORES` to `FALSE` will not reduce the size of Salvo's ecbs. Either configuration option, when `TRUE`, may enlarge the size of ecbs by one byte.

OSBYTES_OF_COUNTS: Set Size of Counters

Name:	`OSBYTES_OF_COUNTS`
Purpose:	To allocate the RAM needed to hold the maximum possible value for counters used in Salvo, and to enable the code to run the counters.
Allowed Values:	`0`, `1`, `2`, `4`
Default Value:	`0`
Action:	If zero, disables all counters. If non-zero, enables the counters `OSctxSws` and `OSidleCtxSws`, and sets the defined type `OStypeCount` to be 8-, 16-, or 32-bit unsigned integer.
Contained in:	`salvo.h`, `rpt.c`
Related:	`OSGATHER_STATISTICS`
Enables:	–
Memory Required:	When non-zero, requires RAM for all enabled counters.

Notes

Salvo uses simple counters to keep track of context switches and notable occurrences. Once a counter reaches its maximum value it remains at that value.

OSBYTES_OF_EVENT_FLAGS: Set Size of Event Flags

Name:	`OSBYTES_OF_EVENT_FLAGS`
Purpose:	To select 8-, 16- or 32-bit event flags.
Allowed Values:	`1`, `2`, `4`
Default Value:	`1`
Action:	Sets the defined type `OStypeEFlag` to 8-, 16- or 32-bit unsigned integer.
Contained in:	`salvo.h`, `flag.c`
Related:	`OSENABLE_EVENT_FLAGS`
Enables:	–
Memory Required:	When event flags are enabled, requires 1, 2 or 4 bytes of RAM for each event flag control block (efcb) and additional ROM (code) dependent on the target processor.

Notes

You can tailor the size of event flags in your Salvo application via this configuration parameter.

Since each bit is independent of the others, it may be to your advantage to have a single, large event flag instead of multiple, smaller ones. For example, the RAM requirements for two 8-bit event flags will exceed those for a single 16-bit event flag since the former requires two event control blocks, whereas the latter needs only one.

OSBYTES_OF_DELAYS: Set Length of Delays

Name:	`OSBYTES_OF_DELAYS`
Purpose:	To enable delays and timeout services and to allocate the RAM needed to hold the maximum specified value (in system ticks) for delays and timeouts.
Allowed Values:	`0`, `1`, `2`, `4`
Default Value:	`0`
Action:	If zero, disables all delay and timeout services. If non-zero, enables the delay and timeout services, and sets the defined type `OStypeDelay` to be 8-, 16- or 32-bit unsigned integer.
Contained in:	`salvo.h`, `rpt.c`
Related:	`OSTIMER_PRESCALAR`
Enables:	`OS_Delay()`, `OSTimer()`
Memory Required:	When non-zero, requires 1, 2 or 4 additional bytes of RAM for each tcb and 1 tcb pointer in RAM.

Notes

Disabling delays and timeouts will reduce the size of the Salvo code considerably. It will also reduce the size of the tcbs by 2 to 6 bytes per tcb.

Use of `OSTIMER_PRESCALAR` in conjunction with `OSBYTES_OF_DELAYS` can provide for very long delays and timeouts while minimizing tcb memory requirements.

OSBYTES_OF_TICKS: Set Maximum System Tick Count

Name:	`OSBYTES_OF_TICKS`
Purpose:	To enable elapsed time services and to allocate the RAM needed to hold the maximum specified system ticks value.
Allowed Values:	`0`, `1`, `2`, `4`
Default Value:	`0`
Action:	If zero, disables all elapsed time services. If non-zero, enables the services , and sets the defined type `OStypeTick` to be 8-, 16- or 32-bit unsigned integer.
Contained in:	`salvo.h`, `rpt.c`
Related:	`OSTIMER_PRESCALAR`
Enables:	`OSGetTicks()`, `OSSetTicks()`, `OSTimer()`
Memory Required:	When non-zero, requires RAM for the system tick counter.

Notes

Salvo uses a simple counter to keep track of system ticks. After it reaches its maximum value the counter rolls over to 0.

Elapsed time services based on the system tick are obtained through `OSGetTicks()` and `OSSetTicks()`.

`OSBYTES_OF_TICKS` must be greater or equal to `OSBYTES_OF_DELAYS`.

OSCALL_OSCREATEEVENT: Manage Interrupts when Creating Events

Name:	`OSCALL_OSCREATEEVENT`
Purpose:	For use on target processors without software stacks in order to manage for interrupts when calling event-creating services.
Allowed Values:	`OSFROM_BACKGROUND`: Your application creates events only in mainline code. `OSFROM_FOREGROUND`: Your application creates events only within interrupts. `OSFROM_ANYWHERE` Your application creates events both in mainline code and within interrupts. You must explicitly control interrupts around `OSCALL_OSCREATEEVENT` (see below).
Default Value:	`OSFROM_BACKGROUND`
Action:	Configures the interrupt control for all Salvo event-creating services.
Contained in:	`salvo.h`, `salvoscg.h`, `multcall.h`
Related:	`OSCALL_OSSIGNALEVENT`, `OSCALL_OSRETURNEVENT`
Enables:	–
Memory Required:	Small variations in ROM depending on its value.

Notes

`OSCALL_OSCREATEEVENT` is required *only* when using a compiler that does not maintain function parameters and auto variables on a software stack or in registers. Therefore this configuration parameter and all similar ones are only needed when using certain target processors and compilers.

Compilers that maintain function parameters and auto variables in a dedicated area of RAM usually do so because a software stack and stack pointers *do not exist* on the target processor. In order to minimize RAM usage, these compilers[54] *overlay* the parameter and variable areas of multiple functions as long as the functions do not occupy the same *call graph*. This is all done transparently – no user involvement is required.

The issue is complicated by wanting to call Salvo services from both mainline (background) and interrupt (foreground) code. In

[54] E.g. the HI-TECH PICC and V8C compilers.

this case, each service needs its own parameter and auto variable area separate from that of mainline-only services, and the user must "wrap" each mainline service with calls to disable and then re-enable interrupts[55] in order to avoid data corruption. See the examples below.

The control of interrupts in each event-creating service like `OSCreateBinSem()` depends on where it is called in your application. In Figure 47 interrupts will be disabled and re-enabled inside `OSCreateBinSem()`. This is referred to as *protecting a critical region of code*, and is typical of RTOS services. In this situation, `OSCALL_OSCREATEEVENT` must be set to `OSFROM_BACKGROUND`.

```
int main( void )
{
  …
  OSCreateBinSem(BINSEM1_P);
  …
}
```

Figure 47: How to call OSCreateBinSem() when OSCALL_OSCREATEEVENT is set to OSFROM_BACKGROUND

In Figure 48 `OSCreateBinSem()` must not change the processor's interrupt status, because re-enabling interrupts within an ISR can cause unwanted nested interrupts. In this situation, set `OSCALL_OSCREATEEVENT` to `OSFROM_FOREGROUND`.

```
interrupt myISR( void )
{
  …
  if ( some_condition )
    OSCreateBinSem(BINSEM2_P);
  …
}
```

Figure 48: How to call OSCreateBinSem() when OSCALL_OSCREATEBINSEM is set to OSFROM_FOREGROUND

In Figure 49, `OSCreateBinSem()` is called from the background as well as the foreground. In this situation, `OSCALL_OSCREATEEVENT` must be set to `OSFROM_ANYWHERE` and `OSCreateBinSem()` must be preceded by `OSProtect()` and followed by `OSUnprotect()` wherever it's called in mainline (background) code.

```
int main( void )
{
```

[55] See "Interrupt Levels" in the HI-TECH PICC and PICC-18 User's Guide.

```
  …
  OSProtect();
  OSCreateBinSem(BINSEM1_P);
  OSUnprotect();
  …
  OSProtect();
  OSCreateBinSem(BINSEM2_P);
  OSUnprotect();
  …
}

interrupt myISR( void )
{
  …
  if ( some_condition )
    OSCreateBinSem(BINSEM2_P);
  …
}
```

Figure 49: How to call OSCreateBinSem() when OSCALL_CREATEBINSEM is set to OSFROM_ANYWHERE

Failing to set `OSCALL_OSCREATEEVENT` properly to reflect where you are calling `OSCreateBinSem()` in your application may cause unpredictable results, and may also result in compiler errors.

With some compilers (e.g. HI-TECH PICC), `OSCALL_OSCREATEEVENT` also automatically enables certain special directives[56] in the Salvo source code to ensure proper compilation.

[56] E.g. `#pragma interrupt_level 0`, to allow a function to be called both from mainline code and from an interrupt. In this situation a function has "multiple call graphs."

OSCALL_OSGETPRIOTASK: Manage Interrupts when Returning a Task's Priority

`OSCALL_OSGETPRIOTASK` manages how interrupts are controlled in `OSGetPrio()` and `OSGetPrioTask()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for services that can be called from the foreground.

OSCALL_OSGETSTATETASK: Manage Interrupts when Returning a Task's State

`OSCALL_OSGETSTATETASK` manages how interrupts are controlled in `OSGetState()` and `OSGetStateTask()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for services that can be called from the foreground.

OSCALL_OSMSGQCOUNT: Manage Interrupts when Returning Number of Messages in Message Queue

`OSCALL_OSMSGQCOUNT` manages how interrupts are controlled in `OSMsgQCount()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for services that can be called from the foreground.

OSCALL_OSMSGQEMPTY: Manage Interrupts when Checking if Message Queue is Empty

`OSCALL_OSMSGQEMPTY` manages how interrupts are controlled in `OSMsgQEmpty()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for services that can be called from the foreground.

OSCALL_OSRETURNEVENT: Manage Interrupts when Reading and/or Trying Events

`OSCALL_OSRETURNEVENT` manages how interrupts are controlled in event-reading and event-trying services (e.g. `OSReadEFlag()` and `OSTrySem()`, respectively).

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for event-reading and event-trying services.

OSCALL_OSSIGNALEVENT: Manage Interrupts when Signaling Events and Manipulating Event Flags

`OSCALL_OSSIGNALEVENT` manages how interrupts are controlled in event-signaling services (e.g. `OSSignalMsg()`), `OSClrEFlag()` and `OSSetEFlag()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for event-signaling services.

OSCALL_OSSTARTTASK: Manage Interrupts when Starting Tasks

`OSCALL_OSSTARTTASK` manages how interrupts are controlled in `OSStartTask()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for event-signaling services.

OSCLEAR_GLOBALS: Explicitly Clear all Global Parameters

Name:	OSCLEAR_GLOBALS
Purpose:	To guarantee that all global variables used by Salvo are explicitly initialized to zero.
Allowed Values:	FALSE, TRUE
Default Value:	TRUE
Action:	If TRUE, configures OSInit() to explicitly fill all global variables (e.g. queue pointers, tcbs, ecbs, etc.) with 0.
Contained in:	salvo.h, init.c, util.c
Related:	OSENABLE_EVENTS, OSENABLE_STACK_CHECKING
Enables:	OSInitTcb() and OSInitEcb() for some values of OSCOMPILER.
Memory Required:	When TRUE, requires a small amount of ROM.

Notes

All ANSI C compilers must initialize global variables to zero. OSInit() clears Salvo's variables by default. For those applications where ROM memory is extremely precious, this configuration option can be disabled, and your application may shrink somewhat as a result.

Caution If you disable this configuration option you must be absolutely sure that your compiler explicitly initializes all of Salvo's global variables to zero. Otherwise your application may not work properly. Even if your compiler does zero all global variables, keep in mind that OSInit() will no longer (re-)zero the global variables, and you will not be able to re-initialize Salvo via a call to OSInit().

OSCLEAR_UNUSED_POINTERS: Reset Unused Tcb and Ecb Pointers

Name:	OSCLEAR_UNUSED_POINTERS
Purpose:	To aid in debugging Salvo activity.
Allowed Values:	FALSE: Salvo makes no attempt to reset no-longer used pointers in tcbs and ecbs. TRUE: Salvo resets all unused tcb and ecb pointers to NULL.
Default Value:	FALSE
Action:	When TRUE, enables code to null unused tcb and ecb pointers.
Contained in:	salvo.h, qdel.c, sched.c
Related:	OSBYTES_OF_DELAYS, OSENABLE_TIMEOUTS,
Enables:	–
Memory Required:	When TRUE, requires a small amount of ROM.

Notes

This configuration option is primarily of use to you if you are interested in viewing or debugging Salvo internals. It is much easier to understand the status of the queues, tasks and events if the unused pointers are NULLed.

Enabling this configuration option will add a few instructions to certain Salvo services.

OSCLEAR_WATCHDOG_TIMER(): Define Instruction(s) to Clear the Watchdog Timer

Name:	`OSCLEAR_WATCHDOG_TIMER`
Purpose:	To clear the processor's watchdog timer within `OSSched()`.
Allowed Values:	Undefined, or defined to be the instruction(s) required to clear the watchdog timer on the target processor.
Default Value:	Undefined
Action:	Each call to `OSSched()` will result the watchdog timer being cleared after the current most-eligible task is dispatched.
Contained in:	`salvo.h`, `sched.c`
Related:	–
Enables:	–
Memory Required:	When defined, requires a small amount of ROM.

Notes

Some processors provide a watchdog timer that generates an internal reset if not cleared within the specified time period. This is used to recover from runaway code. It is generally good coding practice to clear the watchdog timer in only one place in your program. The watchdog timer is often cleared with a single instruction.

Since a Salvo application calls the scheduler repeatedly, an excellent place to clear the watchdog timer is from within `OSSched()`. For example, to clear the watchdog timer from within `OSSched()` on a midrange Microchip PIC processor when using the HI-TECH PICC compiler, add the following line to `salvocfg.h`:

```
#define OSCLEAR_WATCHDOG_TIMER() asm("clrwdt")
```

With the watchdog timer running and cleared from within `OSSched()`, if a task in your application ever fails to yield back to the scheduler, the watchdog timer will expire and generate a watchdog reset.

OSCOMBINE_EVENT_SERVICES: Combine Common Event Service Code

Name:	`OSCOMBINE_EVENT_SERVICES`
Purpose:	To minimize code size with multiple event types enabled.
Allowed Values:	`FALSE`: All event services are implemented as separate, independent functions. `TRUE`: Event services use common code where possible.
Default Value:	`FALSE`
Action:	Changes the structure of the Salvo source code to produce minimum aggregate or individual size of event services.
Contained in:	`salvo.h`, `binsem.c`, `event.c`, `flag.c`, `msg.c`, `msgq.c`, `sem.c`
Related:	–
Enables:	–
Memory Required:	When `TRUE`, reduces ROM requirements when event services for two or more event types are used.

Notes

The services for creating, signaling and waiting events contain common source code. When `OSCOMBINE_EVENT_SERVICES` is `TRUE`, event services use that common code, e.g. `OSCreateBinSem()` and `OSCreateMsgQ()` use the same underlying function. This means that the incremental increase in size of the object code is relatively small when another event type is enabled via `OSENABLE_XYZ`.

When `OSCOMBINE_EVENT_SERVICES` is `FALSE`, each event service is implemented as a separate, independent function, and some code is therefore duplicated. This is used when generating the Salvo freeware libraries for maximum versatility.

When creating an application using two or more event types, the aggregate size of all of the event services will be smaller when `OSCOMBINE_EVENT_SERVICES` is `TRUE`.

The C language `va_arg()` and related functions are required when `OSCOMBINE_EVENT_SERVICES` is `TRUE`.

Setting `OSCOMBINE_EVENT_SERVICES` to `TRUE` with HI-TECH 8051C and the small or medium memory models will prevent you

from calling any allowed event services (e.g. `OSSignalMsg()`) from an ISR. This restriction is lifted in the large model.

OSCTXSW_METHOD: Identify Context-Switching Methodology in Use

Name:	`OSCTXSW_METHOD`
Purpose:	To configure the inner workings of the Salvo context switcher.
Allowed Values:	`OSRTNADDR_IS_PARAM`: `OSSaveRtnAddr()` is passed the task's return address as a parameter. `OSRTNADDR_IS_VAR`: `OSSaveRtnAddr()` reads the tasks's return address through a global variable. `OSVIA_OSCTXSW`: `OSCtxSw()` is used to return to the scheduler. `OSVIA_OSDISPATCH`: `OSCtxSw()` is used in conjunction with `OSDispatch()`.
Default Value:	Defined for each compiler and target in `portXyz.h`. If left undefined, default is `OSRTNADDR_IS_PARAM`.
Action:	Configures Salvo source code for use with the selected compiler and target processor.
Contained in:	`portXyz.h`, `salvo.h`, `init.c`, `mem.c`, `util.c`
Related:	`OSRTNADDR_OFFSET`
Enables:	–
Memory Required:	When set to `OSRTNADDR_IS_VAR`, requires a small amount of RAM. ROM requirements vary.

Notes

This configuration option is used within the Salvo source code to implement part of the context switcher `OS_Yield()`.

Warning Unless you are porting Salvo to an as-yet-unsupported compiler, do not override the value of `OSCTXSW_METHOD` in the porting file `portXyz.h` appropriate for your compiler. Unpredictable results will occur.

If you are working with an as-yet-unsupported compiler, refer to the Salvo source code and *Chapter 10 • Porting* for further instructions.

OSDISABLE_ERROR_CHECKING: Disable Runtime Error Checking

Name:	`OSDISABLE_ERROR_CHECKING`
Purpose:	To turn off runtime error checking.
Allowed Values:	`FALSE`: Error checking is enabled. `TRUE`: Error checking is disabled.
Default Value:	`FALSE`
Action:	Disables certain error checking in some Salvo user services.
Contained in:	`salvo.h`, `binsem.c`, `event.c`, `init.c`, `msg.c`, `msgq.c`, `prio.c`, `sem.c`, `start.c`, `timer.c`, `util.c`
Related:	–
Enables:	–
Memory Required:	When `FALSE`, requires ROM for error-checking.

Notes

By default, Salvo performs run-time error checking on certain parameters passed to user services, like task priorities.

This error checking can be costly in terms of code space (ROM) used. It can be disabled by setting `OSDISABLE_ERROR_CHECKING` to `TRUE`. However, this is never recommended.

Caution Disabling error checking is strongly discouraged. It should only be used as a last resort in an attempt to shrink code size, with the attendant knowledge that any run-time error that goes unchecked may result in unpredictable behavior.

OSDISABLE_FAST_SCHEDULING: Configure Round-Robin Scheduling

Name:	`OSDISABLE_FAST_SCHEDULING`
Purpose:	To alter execution sequence of tasks running in a round-robin manner.
Allowed Values:	`FALSE`: Fast scheduling is used. `TRUE`: Fast scheduling is not used.
Default Value:	`FALSE`
Action:	Changes the way in which eligible tasks returning to the scheduler are re-enqueued into the eligible queue.
Contained in:	`salvo.h`, `sched.c`
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of additional ROM.

Notes

By default, the Salvo scheduler immediately re-enqueues the current task upon its return to the scheduler if it is still eligible. This has a side effect on round-robin scheduling that is best illustrated by example.

If `OSDISABLE_FAST_SCHEDULING` is `FALSE` and the current task signals an event upon which another task of equal priority is waiting, then the scheduler will run the signaling task again *before* the waiting task.[57] On the other hand, if `OSDISABLE_FAST_SCHEDULING` is `TRUE` in this situation, then the scheduler will run the waiting task before the signaling task. In other words, the round-robin sequence of task execution matches the order in which the tasks are made eligible if `OSDISABLE_FAST_SCHEDULING` is set to `TRUE`.

Setting `OSDISABLE_FAST_SCHEDULING` to `TRUE` will have a small but significant negative impact on the context-switching speed of your application.

[57] This is indirectly related to the minimal stack depth required by `OSSignalXyz()` services.

OSDISABLE_TASK_PRIORITIES: Force All Tasks to Same Priority

Name:	`OSDISABLE_TASK_PRIORITIES`
Purpose:	To reduce code (ROM) size when an application does not require prioritized tasks.
Allowed Values:	`FALSE`: Tasks can have assigned priorities. `TRUE`: All tasks have same priority (0).
Default Value:	`FALSE`
Action:	Removes priority-setting and priority-dependent code from Salvo services.
Contained in:	`salvo.h`, `init.c`, `prio.c`, `qins.c`, `task.c`
Related:	–
Enables:	–
Memory Required:	When `FALSE`, requires ROM for management of task priorities.

Notes

By default, Salvo schedules task execution based on task priorities. Some savings in ROM size can be realized by disabling Salvo's priority-specific code. When `OSDISABLE_TASK_PRIORITIES` is set to `TRUE`, all tasks run at the same priority and round-robin.

OSENABLE_BINARY_SEMAPHORES: Enable Support for Binary Semaphores

Name:	`OSENABLE_BINARY_SEMAPHORES`
Purpose:	To control compilation of binary semaphore code via the preprocessor.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, binary semaphore services are not available. If `TRUE`, `OSCreateBinSem()`, `OSSignalBinSem()`and `OS_WaitBinSem()` are available.
Contained in:	`salvo.h`, `binsem.c`, `event.c`, `mem.c`
Related:	`OSENABLE_EVENT_FLAGS`, `OSENABLE_MESSAGES`, `OSENABLE_MESSAGE_QUEUES`, `OSENABLE_SEMAPHORES`, `OSEVENTS`
Enables:	–
Memory Required:	When `TRUE`, requires ROM for binary semaphore services.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to `binsem.c` in your source code, you can control its compilation solely via this configuration option in `salvocfg.h`. This may be more convenient than, say, editing your source code or modifying your project.

OSENABLE_BOUNDS_CHECKING: Enable Runtime Pointer Bounds Checking

Name:	OSENABLE_BOUNDS_CHECKING
Purpose:	To check for out-of-range pointer arguments.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, pointer arguments are not bounds-checked. If TRUE, some services return an error if the pointer argument is out-of-bounds.
Contained in:	salvo.h, binsem.c, eflag.c, event.c, init.c, msg.c, msgq.c, sem.c, task.c
Related:	OSDISABLE_ERROR_CHECKING, OSSET_LIMITS
Enables:	–
Memory Required:	When TRUE, requires ROM for pointer bounds checking.

Notes

The result of passing an incorrect pointer to a service is unpredictable. Some protection can be achieved by bounds-checking the pointer to ensure that it is within a valid range of pointer values appropriate for the service. This can be useful when debugging an application that uses variables as placeholders for pointers instead of constants.

The utility of runtime pointer bounds checking is limited. Since valid pointers do not have successive addresses, the allowed range includes not only the valid pointer values but also all the other values within that range. Therefore runtime pointer bounds checking will only detect a small subset of invalid pointer arguments.

OSENABLE_BOUNDS_CHECKING is overridden (i.e. set to TRUE) when OSSET_LIMITS is set to TRUE.

OSENABLE_EVENT_FLAGS: Enable Support for Event Flags

Name:	OSENABLE_EVENT_FLAGS
Purpose:	To control compilation of event flag code via the preprocessor.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, event flag services are not available. If TRUE, OSCreateEFlag(), OSClrEFlag(), OSSetEFlag()and OS_WaitEFlag() are available.
Contained in:	salvo.h, event.c, flag.c, mem.c
Related:	OSBYTES_OF_EVENT_FLAGS, OSENABLE_BINARY_SEMAPHORES, OSENABLE_MESSAGES, OSENABLE_MESSAGE_QUEUES, OSENABLE_SEMAPHORES, OSEVENTS, OSEVENT_FLAGS
Enables:	–
Memory Required:	When TRUE, requires ROM for event flag services.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to eFlag.c in your source code, you can control its compilation solely via this configuration option in salvocfg.h. This may be more convenient than, say, editing your source code or modifying your project.

A value of 0 for OSEVENT_FLAGS automatically resets (overrides) OSENABLE_EVENT_FLAGS to FALSE.

OSENABLE_EVENT_READING: Enable Support for Event Reading

Name:	OSENABLE_EVENT_READING
Purpose:	To control compilation of event-reading code via the preprocessor.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, event-reading services are not available. If TRUE, OSReadBinSem(), OSReadEFlag(), OSReadMsg(), OSReadMsgQ()and OSReadSem() are available.
Contained in:	salvo.h, binsem.c, eflag.c, msg.c, msgq.c, sem.c
Related:	OSCALL_OSRETURNEVENT, OSENABLE_EVENT_TRYING
Enables:	–
Memory Required:	When TRUE, requires ROM for event-reading services.

Notes

If you use any event-reading services (e.g. OSReadMsg()), you must set OSENABLE_EVENT_READING to TRUE in salvocfg.h. If you do not use any event-reading services, leave it at is default value of FALSE.

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including Salvo event source code in your project, you can keep unused event-reading services out of your final object file solely via this configuration option in salvocfg.h. This may be more convenient than, say, editing your source code or modifying your project.

A value of TRUE for OSENABLE_EVENT_TRYING automatically sets (overrides) OSENABLE_EVENT_READING to TRUE.

OSENABLE_EVENT_TRYING: Enable Support for Event Trying

Name:	OSENABLE_EVENT_TRYING
Purpose:	To control compilation of event-trying code via the preprocessor.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, event-trying services are not available. If TRUE, OSTryBinSem(), OSTryMsg(), OSTryMsgQ()and OSTry-Sem() are available.
Contained in:	salvo.h, binsem.c, eflag.c, msg.c, msgq.c, sem.c
Related:	OSCALL_OSRETURNEVENT, OSENABLE_EVENT_READING
Enables:	–
Memory Required:	When TRUE, requires ROM for event-trying services.

Notes

If you use any event-trying services (e.g. OSTrySem()), you must set OSENABLE_EVENT_TRYING to TRUE in salvocfg.h. If you do not use any event-trying services, leave it at is default value of FALSE.

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including Salvo event source code in your project, you can keep unused event-trying services out of your final object file solely via this configuration option in salvocfg.h. This may be more convenient than, say, editing your source code or modifying your project.

A value of TRUE for OSENABLE_EVENT_TRYING automatically sets (overrides) OSENABLE_EVENT_READING to TRUE.

OSENABLE_FAST_SIGNALING: Enable Fast Event Signaling

Name:	`OSENABLE_FAST_SIGNALING`
Purpose:	To increase the rate at which events can be signaled.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, signaled events are processed[58] when the waiting task runs. If `TRUE`, signaled events are processed when the event is signaled.
Contained in:	`salvo.h`, `binsem.c`, `event.c`, `msg.c`, `msgq.c`, `sched.c`, `sem.c`
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires a moderate amount of additional ROM, and extra tcb RAM for messages and message queues.

Notes

With `OSENABLE_FAST_SIGNALING` set to `FALSE`, when an event is signaled and a task was waiting the event, the event remains signaled until the waiting task runs. For example, when a binary semaphore is signaled with `TaskA()` waiting, `OSSignalBinSem()` will return `OSERR_EVENT_FULL` if called again before `TaskA()` runs. When `TaskA()` runs, the binary semaphore is reset to 0, and a subsequent call to `OSSignalBinSem()` will succeed. On the other hand, if `OSENABLE_FAST_SIGNALING` is `TRUE`, the binary semaphore will immediately return to zero when `TaskA()` is made eligible by `OSSignalBinSem()`, and thereafter the binary semaphore can be signaled again without error.

Fast signaling is useful when multiple tasks are waiting an event, or the same event is signaled in rapid succession. In these situations, `OSSignalXyz()` will succeed until no tasks are waiting the event and the event has been signaled.

[58] E.g. a semaphore is decremented.

OSENABLE_IDLE_COUNTER: Track Scheduler Idling

Name:	OSENABLE_IDLE_COUNTER
Purpose:	To count how many times the scheduler has been idle.
Allowed Values:	FALSE: Salvo does not keep track of how often the scheduler OSSched() is idle. TRUE: The OSidleCtxSw counter is incremented each time the scheduler is called with no eligible tasks, i.e. the system is idle.
Default Value:	FALSE
Action:	If TRUE, configures Salvo to track scheduler idling.
Contained in:	salvo.h, init.c, mem.c, rpt.c
Related:	OSGATHER_STATISTICS, OSENABLE_IDLING_HOOK
Enables:	–
Memory Required:	When TRUE, requires a small amount of ROM, plus one byte of RAM.

Notes

If OSGATHER_STATISTICS, OSENABLE_COUNTS and OSENABLE_IDLE_COUNTER are all TRUE, and Salvo's idling hook function is enabled via OSENABLE_IDLING_HOOK, then the OSidleCtxSws counter will be incremented each time the scheduler is called and there are no tasks eligible to run. The percentage of time your application is spending idle can be obtained by:

$$\text{idle time} = (\texttt{OSidleCtxSws} \ / \ \texttt{OSctxSws}) \times 100$$

OSENABLE_IDLING_HOOK: Call a User Function when Idling

Name:	`OSENABLE_IDLING_HOOK`
Purpose:	To provide a simple way of calling a user function when idling.
Allowed Values:	`FALSE`: No user function is called when idling. `TRUE`: An external function named `OSIdlingHook()` is called when idling.
Default Value:	`FALSE`
Action:	If `TRUE`, `OSSched()` calls `OSIdlingHook()` when no tasks are eligible to run.
Contained in:	`salvo.h`,
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM.

Notes

When you enable this both configuration, you must also define an external function `void OSIdlingHook(void)`. It will be called automatically when your Salvo application is idling.

OSENABLE_INTERRUPT_HOOKS: Call User Functions when Controlling Interrupts

Name:	`OSENABLE_INTERRUPT_HOOKS`
Purpose:	To provide a simple way of calling a pair of user functions when Salvo disables and enables interrupts.
Allowed Values:	`FALSE`: No user function are called from `OSDi()` and `OSEi()`, respectively. `TRUE`: An external, user-supplied function named `OSDisableIntsHook()` is called by `OSDi()` *after* interrupts are disabled, and another such function called `OSEnableIntsHook()` is called by `OSEi()` *before* interrupts are enabled.
Default Value:	`FALSE`
Action:	If `TRUE`, you must define your own functions to be called automatically each time Salvo controls interrupts.
Contained in:	`salvo.h`
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires a moderate amount of ROM and some RAM.

Notes

This configuration option is provided as part of a user-defined mechanism for characterizing how long interrupts are disabled by Salvo during runtime.

If your application has a means of counting instruction cycles (e.g. through a free-running counter incrementing with each instruction executed), you can obtain the number of instruction cycles during which interrupts are disabled by writing two user-defined functions.

For example, you could write `OSDisableIntsHook()` to read the instruction cycle counter and store its value in a global variable. `OSEnableIntsHook()` would then read the counter, subtract the global variable from it, and compares it against another global variable used to store a maximum value.

Both `OSDisableIntsHook()` and `OSEnableIntsHook()` run while interrupts are disabled. Their overhead (in instruction cycles) must

be taken into account when characterizing the duration of interrupts being disabled.

OSENABLE_MESSAGES: Enable Support for Messages

Name:	OSENABLE_MESSAGES
Purpose:	To control compilation of message code via the preprocessor.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, message services are not available. If TRUE, OSCreateMsg(), OSSignalMsg() and OS_WaitMsg() are available.
Contained in:	salvo.h, event.c, mem.c, msg.c
Related:	OSENABLE_BINARY_SEMAPHORES, OSENABLE_EVENT_FLAGS, OSENABLE_MESSAGE_QUEUES, OSENABLE_SEMAPHORES, OSEVENTS
Enables:	–
Memory Required:	When TRUE, requires ROM for message services.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to msg.c in your source code, you can control its compilation solely via this configuration option in salvocfg.h. This may be more convenient than, say, editing your source code or modifying your project.

OSENABLE_MESSAGE_QUEUES: Enable Support for Message Queues

Name:	OSENABLE_MESSAGE_QUEUES
Purpose:	To control compilation of message queue code via the preprocessor.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, message services are not available. If TRUE, OSCreateMsgQ(), OSSignalMsgQ() and OS_WaitMsgQ() are available.
Contained in:	salvo.h, event.c, mem.c, msgq.c
Related:	OSENABLE_BINARY_SEMAPHORES, OSENABLE_EVENT_FLAGS, OSENABLE_MESSAGES, OSENABLE_SEMAPHORES, OSEVENTS, OSMESSAGE_QUEUES
Enables:	–
Memory Required:	When TRUE, requires ROM for message queue services.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to msgq.c in your source code, you can control its compilation solely via this configuration option in salvocfg.h. This may be more convenient than, say, editing your source code or modifying your project.

A value of 0 for OSMESSAGE_QUEUES automatically resets (overrides) OSENABLE_MESSAGE_QUEUES to FALSE.

OSENABLE_SCHEDULER_HOOK: Call User Function Inside Scheduler

Name:	`OSENABLE_SCHEDULER_HOOK`
Purpose:	To provide a simple way of calling a user function from inside the scheduler.
Allowed Values:	`FALSE`: No user function is called from `OSSched()`. `TRUE`: An external, user-supplied function named `OSSchedHook()`is called within `OSSched()` *after* events and delays are processed, but *before* the most eligible task is dispatched.
Default Value:	`FALSE`
Action:	If `TRUE`, you must define your own function to be called automatically each time the scheduler runs.
Contained in:	`salvo.h`, `sched.c`
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires ROM for user function and function call.

Notes

This configuration option is provided for advanced users who want to call a function immediately prior to the most eligible task being dispatched by the scheduler.

OSENABLE_SEMAPHORES: Enable Support for Semaphores

Name:	OSENABLE_SEMAPHORES
Purpose:	To control compilation of semaphore code via the preprocessor.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, semaphore services are not available. If TRUE, OSCreateSem(), OSSignalSem() and OS_WaitSem() are available.
Contained in:	salvo.h, event.c, mem.c, sem.c
Related:	OSENABLE_BINARY_SEMAPHORES, OSENABLE_EVENT_FLAGS, OSENABLE_MESSAGES, OSENABLE_MESSAGE_QUEUES, OSEVENTS
Enables:	–
Memory Required:	When TRUE, requires ROM for semaphore services.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to sem.c in your source code, you can control its compilation solely via this configuration option in salvocfg.h. This may be more convenient than, say, editing your source code or modifying your project.

OSENABLE_STACK_CHECKING: Monitor Call ... Return Stack Depth

Name:	OSENABLE_STACK_CHECKING
Purpose:	To enable the user to discern the maximum call ... return stack depth used by Salvo services.
Allowed Values:	FALSE: Stack depth checking is not performed. TRUE: Maximum and current stack depth is recorded.
Default Value:	FALSE
Action:	If TRUE, enables code in each function to monitor the current call ... return stack depth and record a maximum call ... return stack depth if it has changed.
Contained in:	salvo.h, binsem.c, chk.c, debug.c, delay.c, destroy.c, event.c, init.c, msg.c, msgq.c, prio.c, qdel.c, qins.c, rpt.c, sched.c, sem.c, stop.c, tick.c, timer.c, util.c, util2.c
Related:	OSGATHER_STATISTICS, OSRpt()
Enables:	–
Memory Required:	When TRUE, requires a considerable amount of ROM, plus two bytes of RAM.

Notes

Current and maximum stack depth are tracked to a maximum call ... return depth of 255.

Current stack depth is held in OSstkDepth. Maximum stack depth is held in OSmaxStkDepth.

Stack depth is only calculated for call ... returns within Salvo code and is not necessarily equal to the current hardware stack depth of your processor. However, for most applications they will be the same since OSSched() is usually called from main().

OSENABLE_TCBEXT0|1|2|3|4|5: Enable Tcb Extensions

Name:	`OSENABLE_TCBEXT0|1|2|3|4|5`
Purpose:	To add user-definable variables to a task's control block.
Allowed Values:	`FALSE`: Named tcb extension is not enabled. `TRUE`: Named tcb extension is enabled.
Default Value:	`FALSE`
Action:	If `TRUE`, creates a user-definable and accessible object of type `OStypeTcbExt0|1|2|3|4|5` within each tcb.
Contained in:	`salvo.h`
Related:	`OSLOC_TCB`, `OSTYPE_TCBEXT0|1|2|3|4|5`, `OScTcbExt0|1|2|3|4|5`, `OStcbExt0|1|2|3|4|5`
Enables:	`tcbExt0|1|2|3|4|5` fields
Memory Required:	When `TRUE`, requires additional RAM per tcb.

Notes

Salvo's standard tcb fields are reserved for the management of tasks and events. In some instances it is useful to additional variables that are unique to the particular task. Salvo's *tcb extensions* are ideal for this purpose.

The default type for a tcb extension is `void *` (i.e. a void pointer). A tcb extension's type can be overridden to any type[59] by using the appropriate `OSTYPE_TCBEXT0|1|2|3|4|5` configuration option.

Once enabled via `OSENABLE_TCBEXT0|1|2|3|4|5`, a tcb extension can be accessed through the `OScTcbExt0|1|2|3|4|5` or `OStcbExt0|1|2|3|4|5` macros.

`OSLOC_TCB` controls the storage type of tcb extensions. Tcb extensions are only initialized if / when `OSInitTcb()` is called, or by the compiler's startup code. Any desired mix of the tcb extensions can be enabled.

Consider the case of several identical tasks, all created from a single task function, which run concurrently. Each task is responsible for one of several identical communications channels, each with its

[59] Including structures, etc.

own I/O and buffers. Enable a tcb extension of type pointer-to-`struct`, and initialize it uniquely for each task. At runtime each task runs independently of the others, managing its own communications channel, defined by the `struct`. Since only one task function need be defined, substantial savings in code size can be realized.

The example in Listing 31 illustrates the use of a single, `unsigned-char`-sized tcb extension `tcbExt1` that each of four identical tasks uses as an index into an array of offsets in the 4KB buffer the tasks share.

```
…

const unsigned offset[4] = { 3072,
                             2048,
                             1024,
                             0    };

void TaskBuff( void )
{
  for (;;) {
    printf("Task %d's buffer ",   OStID(OScTcbP));
    printf("starts at %d\n",  offset[OScTcbExt1]);
    …
    OS_Yield(label);
  }
}

main()
{
  OSInit();

  OSCreateTask(TaskBuff, OSTCBP(2), 1);
  OSCreateTask(TaskBuff, OSTCBP(6), 1);
  OSCreateTask(TaskBuff, OSTCBP(7), 1);
  OSCreateTask(TaskBuff, OSTCBP(8), 1);

  OStcbExt1(OSTCBP(2)) = 0;
  OStcbExt1(OSTCBP(6)) = 1;
  OStcbExt1(OSTCBP(7)) = 2;
  OStcbExt1(OSTCBP(8)) = 3;

  for ( i = 0 ; i < 4 ; i++ ) {
    OSSched();
  }
}
```

Listing 31: Tcb Extension Example

Each time `TaskBuff()` runs, it can obtain its offset into the 4KB buffer through `OStcbExt1` for the current task, namely, itself. For this example, `OSENABLE_TCBEXT1` was set to `TRUE` and

OSTYPE_TCBEXT1 was set to unsigned char in the project's salvocfg.h. The resulting output is shown in Figure 50.

```
Task 2's buffer starts at 3072
Task 6's buffer starts at 2048
Task 7's buffer starts at 1024
Task 8's buffer starts at 0
```

Figure 50: Tcb Extension Example Program Output

Tcb extensions can be used for a variety of purposes, including

- Passing information via a pointer to a task at startup or during runtime.[60]
- Avoiding the use of task-specific global variables accessed indirectly via OStID().
- Embedding objects of any type in a task's tcb.

60 This is useful because Salvo tasks must be declared as void Task (void), i.e. without any parameters.

OSENABLE_TIMEOUTS: Enable Support for Timeouts

Name:	`OSENABLE_TIMEOUTS`
Purpose:	To be able to specify an optional timeout when waiting for an event.
Allowed Values:	`FALSE`: Timeouts cannot be specified. `TRUE`: Timeouts can be specified.
Default Value:	`FALSE`
Action:	If `TRUE`, enables the passing of an extra parameter to specify a timeout when waiting for an event..
Contained in:	`salvo.h`, `binsem.c`, `delay.c`, `event.c`, `init.c`, `mem.c`, `msg.c`, `msgq.c`, `qdel.c`, `qins.c`, `rpt.c`, `sched.c`, `sem.c`
Related:	–
Enables:	`OSTimedOut()`
Memory Required:	When `TRUE`, requires a considerable amount of ROM, plus an additional byte of RAM per tcb.

Notes

By specifying a timeout when waiting for an event, the waiting task can continue if the event does not occur within the specified time period. Use `OSTimedOut()` to detect if a timeout occurred.

If timeouts are enabled, you can use the defined symbol `OSNO_TIMEOUT` for those calls that do not require a timeout.

See *Chapter 6 • Frequently Asked Questions (FAQ)* for more information on using timeouts.

OSGATHER_STATISTICS: Collect Run-time Statistics

Name:	`OSGATHER_STATISTICS`
Purpose:	To collect run-time statistics from your application.
Allowed Values:	`FALSE`: Statistics are not collected. `TRUE`: A variety of statistics are collected.
Default Value:	`FALSE`
Action:	If `TRUE`, enables Salvo code to collect run-time statistics from your application on the number of errors, warnings, timeouts, context switches and calls to the idle function.
Contained in:	`salvo.h`, `init.c`, `mem.c`, `rpt.c`, `sched.c`
Related:	`OSBYTES_OF_COUNTS`, `OSENABLE_STACK_CHECKING`
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM, plus RAM for counters.

Notes

The numbers of errors, warnings and timeouts are tracked to a maximum value of 255.

The maximum number of any counter is dependent on the value of `OSBYTES_OF_COUNTS`. If `OSBYTES_OF_COUNTS` is not defined or is defined to be 0, it will be redefined to 1.

Which statistics are collected is highly dependent on the related configuration options listed above.

If enabled via `OSLOGGING`, error and warning logging will occur regardless of the value of `OSGATHER_STATISTICS`.

OSINTERRUPT_LEVEL: Specify Interrupt Level for Interrupt-callable Services

Name:	`OSINTERRUPT_LEVEL`
Purpose:	To specify the interrupt level used in the Salvo source code. For use with these compilers: HI-TECH PICC and PICC-Lite HI-TECH PICC-18 HI-TECH V8C
Allowed Values:	`0-7` (depends on compiler)
Default Value:	`0`
Action:	
Contained in:	`salvo.h`, `salvolvl.h`
Related:	`OSCALL_OSXYZ`
Enables:	–
Memory Required:	–

Notes

Some compilers support an interrupt level feature. With `OSINTERRUPT_LEVEL` you can specify which level is used by Salvo services called from the foreground.

All affected Salvo services use the same interrupt level.

OSLOC_ALL: Storage Type for All Salvo Objects

Name:	OSLOC_ALL
Purpose:	To place Salvo objects anywhere in RAM.
Allowed Values:	See Table 5.
Default Value:	OSLOC_DEFAULT (in portxyz.h).
Action:	Set the memory storage type for all of Salvo's objects that aren't overridden by OSLOC_XYZ.
Contained in:	salvo.h
Related:	OSLOC_ALL, OSLOC_COUNT, OSLOC_CTCB, OSLOC_DEPTH, OSLOC_ECB, OSLOC_ERR, OSLOC_LOGMSG, OSLOC_MQCB, OSLOC_MSGQ, OSLOC_PS, OSLOC_SIGQ, OSLOC_TCB, OSLOC_TICK
Enables:	–
Memory Required:	n/a

Notes

Many compilers support a variety of storage types (also called *memory types*) for static objects. Depending on the target processor's architecture, it may be advantageous or necessary to place Salvo's variables into RAM spaces other than the default provided by the compiler.

OSLOC_ALL, when used alone, will locate all of Salvo's objects in the specified RAM space. OSLOC_ALL overrides all other undefined OSLOC_XYZ configuration parameters. To place all of Salvo's variables in RAM Bank 2 with the HI-TECH PICC compiler, use:

```
#define OSLOC_ALL bank2
```

in salvocfg.h. To place the event control blocks (ecbs) in data RAM, and everything else in external RAM with the Keil C51 compiler, use:

```
#define OSLOC_ALL xdata
#define OSLOC_ECB data
```

The storage types for *all* of Salvo's objects are set via OSLOC_ALL and the remaining OSLOC_XYZ (see below) configuration parameters. *Do not attempt to set storage types in any other manner* – compile- and / or run-time errors are certain to result.

Table 5 lists the allowable storage types / type qualifiers for Salvo objects for each supported compiler (where applicable). Those on separate lines can be combined, usually in any order.

compiler	storage types / type qualifiers
HI-TECH PICC	`bank1`, `bank2`, `bank3` `persistent`
HI-TECH PICC-18	`near` `persistent`
HI-TECH V8C	`persistent`
Keil C51	`data`, `idata`, `xdata`
Microchip MPLAB-C18	not supported – use `OSMPLAB_C18_LOC_ALL_NEAR` instead

Table 5: Allowable Storage Types / Type Qualifiers for Salvo Objects

See Also `OSLOC_XYZ`, *Chapter 11 • Tips, Tricks and Troubleshooting*

OSLOC_COUNT: Storage Type for Counters

Name:	OSLOC_COUNT
Purpose:	To place Salvo counters anywhere in RAM.
Allowed Values:	See Table 5.
Default Value:	OSLOC_DEFAULT (in portxyz.h).
Action:	Set storage type for Salvo counters.
Contained in:	salvo.h, mem.c
Related:	OSLOC_ALL
Enables:	–
Memory Required:	n/a

Notes

OSLOC_COUNT will locate the context switch and idle context switch counters in the specified RAM area. Memory is allocated for these counters only when statistics are gathered.

To explicitly specify RAM Bank 0 with the HI-TECH PICC compiler, use:

```
#define OSLOC_COUNT
```

in salvocfg.h.

As with all OSLOC_XYZ configuration options, multiple type qualifiers can be used with OSLOC_COUNT. For example, to prevent HI-TECH PICC start-up code from re-initializing Salvo's counters in RAM bank 2, use:

```
#define OSLOC_COUNT bank2 persistent
```

See Also

Chapter 11 • Tips, Tricks and Troubleshooting

OSLOC_CTCB: Storage Type for Current Task Control Block Pointer

`OSLOC_CTCB` will locate the current task control block pointer in the specified RAM area. This pointer is used by `OSSched()`.

See `OSLOC_COUNT` for more information on setting storage types for Salvo objects.

OSLOC_DEPTH: Storage Type for Stack Depth Counters

`OSLOC_DEPTH` will locate the 8-bit call ... return stack depth and maximum stack depth counters in the specified RAM area. Memory is allocated for these counters only when stack depth checking is enabled.

See `OSLOC_COUNT` for more information on setting storage types for Salvo objects.

See Also `OSENABLE_STACK_CHECKING`

OSLOC_ECB: Storage Type for Event Control Blocks and Queue Pointers

`OSLOC_ECB` will locate the event control blocks, the eligible queue pointer and the delay queue pointer in the specified RAM area. Memory is allocated for ecbs only when events are enabled. Memory is allocated for the delay queue pointer only when delays and/or timeouts are enabled.

See `OSLOC_COUNT` for more information on setting storage types for Salvo objects.

See Also `OSEVENTS`

OSLOC_EFCB: Storage Type for Event Flag Control Blocks

`OSLOC_EFCB` will locate the event flag control blocks – declared to be of type `OSgltypeEfcb` by the user – in the specified RAM area.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

OSLOC_ERR: Storage Type for Error Counters

OSLOC_ERR will locate the 8-bit error, warning and timeout counters in the specified RAM area. Memory is allocated for these counters only when logging is enabled.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OSENABLE_TIMEOUTS, OSGATHER_STATISTICS, OS_LOGGING

OSLOC_GLSTAT: Storage Type for Global Status Bits

OSLOC_GLSTAT will locate Salvo's global status bits in the specified RAM area. Memory is allocated for these bits whenever time functions are enabled.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

OSLOC_LOGMSG: Storage Type for Log Message String

OSLOC_LOGMSG will locate the character buffer used to hold log messages in the specified RAM area. This buffer is needed to create error, warning and descriptive informational messages.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OS_LOGGING, OSLOG_MESSAGES

OSLOC_MQCB: Storage Type for Message Queue Control Blocks

OSLOC_MQCB will locate the message queue control blocks (mqcbs) in the specified RAM area. Each message queue has an mqcb asso-

ciated with it – however, message queues and mqcbs need not be in the same bank.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

OSLOC_MSGQ: Storage Type for Message Queues

OSLOC_MSGQ tells Salvo that the message queue buffers are located in the specified RAM area. By using the predefined Salvo qualified type OSgltypeMsgQP when declaring each buffer it will be automatically placed in the desired RAM bank.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OSMESSAGE_QUEUES

OSLOC_PS: Storage Type for Timer Prescalar

OSLOC_PS will locate the timer prescalar (used by OSTimer()) in the specified RAM area.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OSENABLE_PRESCALAR

OSLOC_TCB: Storage Type for Task Control Blocks

OSLOC_TCB will locate the task control blocks in the specified RAM area.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

OSLOC_SIGQ: Storage Type for Signaled Events Queue Pointers

`OSLOC_SIGQ` will locate the signaled events queue pointers in the specified RAM area. Memory is allocated for this counter only when events are enabled.

See `OSLOC_COUNT` for more information on setting storage types for Salvo objects.

OSLOC_TICK: Storage Type for System Tick Counter

`OSLOC_TICK` will locate the system tick counter in the specified RAM area. Memory is allocated for this counter only when ticks are enabled.

See `OSLOC_COUNT` for more information on setting storage types for Salvo objects.

See Also

`OSBYTES_OF_TICKS`

OSLOGGING: Log Runtime Errors and Warnings

Name:	OSLOGGING
Purpose:	To log runtime errors and warnings.
Allowed Values:	FALSE: Errors and warnings are not logged. TRUE: Errors and warnings are logged.
Default Value:	FALSE
Action:	Configures Salvo functions to log all errors and warnings that occur when during execution.
Contained in:	salvo.h, debug.c, init.c, mem.c, rpt.c, util.c
Related:	OSLOG_MESSAGES, OSRpt()
Enables:	–
Memory Required:	When TRUE, requires a considerable amount of ROM, plus RAM for the error and warning counters.

Notes

Most Salvo functions return an 8-bit error code. Additionally, Salvo can track run-time errors and warnings through the dedicated 8-bit counters OSerrs and OSwarns.

OSRpt() will display the error and warning counters if OSLOGGING is TRUE.

The value of OSLOGGING has no effect on the return codes for Salvo user services.

OSLOGGING is not affected by OSGATHER_STATISTICS.

See Also

OSRpt()

OSLOG_MESSAGES: Configure Runtime Logging Messages

Name:	OSLOG_MESSAGES
Purpose:	To aide in debugging your Salvo application.
Allowed Values:	OSLOG_NONE: No messages are generated. OSLOG_ERRORS: Only error messages are generated. OSLOG_WARNINGS: Error and warning messages are generated. OSLOG_ALL: Error, warning and informational messages are generated.
Default Value:	OSLOG_NONE
Action:	Configures Salvo functions to log in a user-understandable way all errors, warnings and/or general information that occurs when each function executes.
Contained in:	salvo.h, debug.c, mem.c
Related:	OSLOGGING
Enables:	–
Memory Required:	When TRUE, requires a considerable amount of ROM, plus RAM for an 80-character buffer, OSlogMsg[].

Notes

Most Salvo functions return an 8-bit error code. If your application has the ability to printf() to a console, Salvo can be configured via this configuration option to report on errors, warnings and/or general information with descriptive messages. If an error, warning or general event occurs, a descriptive message with the name of the corresponding Salvo function is output via printf(). This can be useful when debugging your application, when modifying the source code or when learning to use Salvo.

Applications that do not have a reentrant printf() may have problems when reporting any errors. In these cases, set OSLOG_MESSAGES to OSLOG_NONE.

Stack depth for printf() is not tracked by Salvo – your application may have problems if there is insufficient stack depth beyond that used by Salvo.

OSLOGGING must be TRUE to use OSLOG_MESSAGES.

The value of `OSLOG_MESSAGES` has no effect on the return codes for Salvo user services.

OSMPLAB_C18_LOC_ALL_NEAR: Locate all Salvo Objects in Access Bank (MPLAB-C18 Only)

Name:	`OSMPLAB_C18_LOC_ALL_NEAR`
Purpose:	To improve application performance by placing Salvo's global objects in access RAM.
Allowed Values:	`FALSE`: Salvo's global objects are placed in banked RAM. `TRUE`: Salvo's global objects are placed in access RAM.
Default Value:	`FALSE`
Action:	Declares all of Salvo's global objects to be of type near.
Contained in:	`salvo.h`, `mem.c`
Related:	–
Enables:	–
Memory Required:	When `TRUE`, should reduce ROM requirements.

Notes

Salvo's `OSLOC_XYZ` configuration cannot be used with MPLAB-C18. Use `OSMPLAB_C18_LOC_ALL_NEAR` instead to place all of Salvo's global objects in access RAM for improved run-time performance.

OSOPTIMIZE_FOR_SPEED: Optimize for Code Size or Speed

Name:	`OSOPTIMIZE_FOR_SPEED`
Purpose:	To allow you to optimize your application for minimum Salvo code size or maximum speed.
Allowed Values:	`FALSE`: Salvo source code will compile for minimum size with existing configuration options. `TRUE`: Salvo source code will compile for maximum speed with existing configuration options.
Default Value:	`FALSE`
Action:	Takes advantage of certain opportunities to increase the speed of the Salvo code.
Contained in:	`salvo.h`, `event.c`, `init.c`, `qdel.c`, `qins.c`, `sched.c`, `util.c`
Related:	`OSENABLE_DELAYS`
Enables:	–
Memory Required:	When `TRUE`, requires small amounts of ROM and RAM.

Notes

Opportunities exist in the Salvo source code to improve execution speed at the cost of some additional lines of code or bytes of RAM. This configuration option enables you to take advantage of these opportunities.

This configuration option does not override other parameters that may also have an effect on code size.

This configuration option is completely independent of any optimizations your compiler may perform. The interaction between it and your compiler is of course unpredictable.

The interplay between execution speed and memory requirements is complex and is most likely to be unique to each application. For example, configuring Salvo for maximum speed may in some cases both increase speed and shrink ROM size, at the expense of some memory RAM.

OSPIC18_INTERRUPT_MASK: Configure PIC18 Interrupt Mode

Name:	`OSPIC18_INTERRUPT_MASK`
Purpose:	To allow you to control which PIC18 PICmicro interrupts are disabled during Salvo's critical sections.
Allowed Values:	`0xC0`, `0x80`, `0x40`, `0x00`
Default Value:	`0xC0` (all interrupts are disabled during critical sections).
Action:	Defines the interrupt-clearing mask that will be used in Salvo services that contain critical regions of code.
Contained in:	`portiar18.h`, `portmcc.h`, `portpic18.c`
Related:	–
Enables:	–
Memory Required:	–

Notes

`OSPIC18_INTERRUPT_MASK` is currently supported for use with the IAR PIC18 and Microchip MPLAB-C18 compilers.

Microchip PIC18 PICmicro MCUs support two distinct interrupt modes of operation: one with two levels of interrupt priorities (`IPEN` is `1`), and one that is compatible with Microchip's mid-range PICmicro devices (`IPEN` is `0`). Depending on how your application calls Salvo services, it may be to your advantage to change `OSPIC18_INTERRUPT_MASK` to minimize interrupt latency.

When `OSPIC18_INTERRUPT_MASK` is set to `0xC0`, all interrupts (global / high-priority and peripheral / low-priority) are disabled during critical regions. Therefore a value of `0xC0` is compatible with both priority schemes and any method of calling Salvo services.

When `OSPIC18_INTERRUPT_MASK` is set to `0x80`, only global / high-priority interrupts are disabled during critical regions. Therefore a value of `0x80` should only be used in two cases: 1) in compatibility mode, and 2) in priority mode if Salvo services that can be called from the foreground / ISR level are called *exclusively from high-level interrupts*.

When `OSPIC18_INTERRUPT_MASK` is set to `0x40`, only peripheral / low-priority interrupts are disabled during critical regions. Therefore a value of `0x40` should only be used in priority mode if Salvo

services that can be called from the foreground / ISR level are called *exclusively from low-level interrupts*. A value of `0x40` must not be used in compatibility mode.

A value of 0x00 is permitted. However, it must only be used on applications that *do not use interrupts*.

Failure to use the correct value of `OSPIC18_INTERRUPT_MASK` for your application will lead to unpredictable runtime results.

See Microchip's PIC18 PICmicro databooks for more information.

OSPRESERVE_INTERRUPT_MASK: Control Interrupt-enabling Behavior

Name:	`OSPRESERVE_INTERRUPT_MASK`
Purpose:	To avoid conflicts arising from Salvo's interrupt control in critical sections.
Allowed Values:	`FALSE`: Interrupts will be unmasked (i.e. enabled) after a critical section. `TRUE`: The interrupt mask will be restored after a critical section.
Default Value:	`TRUE`
Action:	Configures `OSEi()` and `DisableInts()` appropriately.
Contained in:	`portXyz.h`, `salvo.h`
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires small amounts of ROM.

Notes

As with any RTOS, Salvo must disable interrupts during critical sections to avoid data corruption. Blindly disabling interrupts at the start of a critical section and re-enabling them at the end can cause problems in interrupt-sensitive applications. By setting `OSPRESERVE_INTERRUPT_MASK` to `TRUE`, Salvo always restores the interrupt mask to its pre-critical-section value (if supported).

In some cases,[61] ROM can be reduced slightly by following a simpler interrupt-control methodology that blindly re-enables (i.e. unmasks) interrupts after a critical section. Set `OSPRESERVE_INTERRUPT_MASK` to `FALSE` if this is desired.

Refer to your compiler's `portXyz.h` to see if `OSPRESERVE_INTERRUPT_MASK` is supported.

[61] I.e. when the application does not explicitly control interrupts other than to enable them initially, and no Salvo services are called from ISRs.

OSRPT_HIDE_INVALID_POINTERS: OSRpt() Won't Display Invalid Pointers

Name:	OSRPT_HIDE_INVALID_POINTERS
Purpose:	To make the output of OSRpt() more legible.
Allowed Values:	FALSE: All tcb and ecb pointer values will be displayed, regardless of whether or not they are valid. TRUE: Only those pointers which are valid are shown in the monitor.
Default Value:	TRUE
Action:	Configures OSRpt() to show or hide invalid pointers.
Contained in:	salvo.h, rpt.c
Related:	OSRPT_SHOW_ONLY_ACTIVE, OSRPT_SHOW_TOTAL_DELAY
Enables:	–
Memory Required:	When TRUE, requires a small amount of ROM.

Notes

In some cases, the pointer fields of tcbs and ecbs are meaningless. For example, if a task has been destroyed, the pointers in its tcb are invalid. By making OSRPT_HIDE_INVALID_POINTERS TRUE, OSRpt()'s output is simplified by removing unnecessary information. Invalid pointers are displayed as "n/a".

See *Chapter 7 • Reference* for more information on OSRpt().

OSRPT_SHOW_ONLY_ACTIVE: OSRpt() Displays Only Active Task and Event Data

Name:	`OSRPT_SHOW_ONLY_ACTIVE`
Purpose:	To remove unnecessary information from `OSRpt()`'s output.
Allowed Values:	`FALSE`: Show the contents of each tcb and ecb. `TRUE`: Show only the contents of each active tcb and ecb.
Default Value:	`TRUE`
Action:	Configures `OSRpt()` to show only tasks which are not destroyed and events which have already been created.
Contained in:	`salvo.h`, `rpt.c`
Related:	`OSRPT_HIDE_INVALID_POINTERS`, `OSRPT_SHOW_TOTAL_DELAY`
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM.

Notes

By showing neither the tcb contents of tasks in the destroyed state, nor the ecb contents of events which have not yet been created, `OSRpt()`'s output is simplified. However, if you wish to have all the tasks and events displayed by `OSRpt()`, set this configuration option to `FALSE`.

See *Chapter 7 • Reference* for more information on `OSRpt()`.

OSRPT_SHOW_TOTAL_DELAY: OSRpt() Shows the Total Delay in the Delay Queue

Name:	`OSRPT_SHOW_TOTAL_DELAY`
Purpose:	To aid in computing total delay times when viewing `OSRpt()`'s output.
Allowed Values:	`FALSE`: Only individual task delay fields are shown. `TRUE`: The total (cumulative) delay for all the tasks in the delay queue is computed and shown.
Default Value:	`TRUE`
Action:	Configures `OSRpt()` to compute and display the total delay of all delayed tasks.
Contained in:	`salvo.h`, `rpt.c`
Related:	`OSRPT_HIDE_INVALID_POINTERS`, `OSRPT_SHOW_ONLY_ACTIVE`
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM.

Notes

Task delays are stored in the delay queue in an incremental (and not absolute) scheme. When debugging your application it may be useful to be able to see the total delay of all tasks in the delay queue.

See *Chapter 7 • Reference* for more information on `OSRpt()`.

OSRTNADDR_OFFSET: Offset (in bytes) for Context-Switching Saved Return Address

Name:	`OSRTNADDR_OFFSET`
Purpose:	To configure the inner workings of the Salvo context switcher.
Allowed Values:	Any literal.
Default Value:	Defined for each compiler and target in `portXyz.h` whenever `OSCTXSW_METHOD` is `OSRTNADDR_IS_VAR`. If left undefined, default is 0.
Action:	Configures Salvo source code for use with the selected compiler and target processor.
Contained in:	`portXyz.h`, `salvo.h`, `util.c`
Related:	`OSCTXSW_METHOD`
Enables:	–
Memory Required:	n/a

Notes

This configuration option is used within the Salvo source code to implement part of the context switcher `OS_Yield()`.

Warning Unless you are porting Salvo to an as-yet-unsupported compiler, do not override the value of `OSCTXSW_METHOD` in the porting file `portXyz.h` appropriate for your compiler. Unpredictable results will occur.

If you are working with an as-yet-unsupported compiler, refer to the Salvo source code and *Chapter 10 • Porting* for further instructions.

OSSCHED_RETURN_LABEL(): Define Label within OSSched()

Name:	`OSSCHED_RETURN_LABEL`
Purpose:	To define a globally visible label for certain Salvo context switchers.
Allowed Values:	Undefined, or defined to be the instruction(s) required to create a globally visible label.
Default Value:	Defined but valueless.
Action:	Creates a globally visible label for use by the `goto` statement.
Contained in:	`salvo.h`, `sched.c`
Related:	–
Enables:	–
Memory Required:	–

Notes

Salvo context switchers for certain compilers and/or target processors may be implemented with a `goto`-based approach rather than with a `call`-based approach. For those circumstances, a globally visible label within the scheduler `OSSched()` is required. By declaring a label via this configuration parameter, a context switcher will be able to "return" from a task to the appropriate part of the scheduler.

The preferred name for the label is `OSSchedRtn`.

For the Microchip 12-bit PICmicros (e.g. PIC16C57), which have only a 2-level hardware call…return stack, the following is used with the HI-TECH PICC compiler:

```
#define OSSCHED_RETURN_LABEL() { \
        asm("global _OSSchedRtn"); \
        asm("_OSSchedRtn:"); \
        }
```

This creates a globally visible label `OSSchedRtn` that can be jumped to from other parts of the program.

See the various `portxyz.h` compiler- and target-specific porting files for more information.

OSSET_LIMITS: Limit Number of Runtime Salvo Objects

Name:	OSSET_LIMITS
Purpose:	To limit the number of permissible Salvo objects when using the freeware libraries.
Allowed Values:	FALSE: The numbers of Salvo objects are limited only by their definitions in mem.c. TRUE: Salvo services reject operations on Salvo objects that are outside the limits set by the configuration parameters.
Default Value:	FALSE
Action:	Adds run-time bounds-checking on pointer arguments.
Contained in:	salvo.h
Related:	OSENABLE_BOUNDS_CHECKING
Enables:	Bounds-checking code sections in various Salvo services.
Memory Required:	When TRUE, requires some ROM.

Notes

Services involving Salvo objects (e.g. events) normally accept pointer arguments to any valid control blocks. However, when OSSET_LIMITS is TRUE, OSENABLE_BOUNDS_CHECKING is set to TRUE, and these services will only accept pointers that are within the control blocks as specified by configuration parameters (e.g. OSEVENTS) at compile time, and otherwise return an error code.

In other words, if OSSignalXyz() is compiled with OSSET_LIMITS as TRUE and OSEVENTS as 4, passing it an event control block pointer (ecbP) of OSECBP(5) or higher[62] will result in OSSignalXyz() returning an error code of OSERR_BAD_P.

All users should leave this option at its default value.

[62] ecbs are numbered from 1 to OSEVENTS.

OSSPEEDUP_QUEUEING: Speed Up Queue Operations

Name:	`OSSPEEDUP_QUEUEING`
Purpose:	To improve queueing performance.
Allowed Values:	`FALSE`: Use standard queueing algorithm. `TRUE`: Use fast queueing algorithm.
Default Value:	`FALSE`
Action:	Configures queueing routines for fastest performance.
Contained in:	`salvo.h`, `qins.c`
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM and RAM.

Notes

It is possible to improve the speed of certain operations involving queues approximately 25% through the use of local variables in a few of Salvo's internal queueing routines.

Applications with minimal RAM should leave this configuration option at its default value.

See Chapter 9 • Performance for more information on queueing.

OSTIMER_PRESCALAR: Configure Prescalar for OSTimer()

Name:	`OSTIMER_PRESCALAR`
Purpose:	To allow you maximum flexibility in locating `OSTimer()` within your application.
Allowed Values:	`0`, `2` to (2^32)-1.
Default Value:	`0`
Action:	If non-zero, adds code and an 8- to 32-bit countdown timer to `OSTimer()` to implement a prescalar.
Contained in:	`salvo.h`, `init.c`, `timer.c`
Related:	`OSBYTES_OF_DELAYS`, `OSBYTES_OF_TICKS`
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM, plus RAM for the prescalar.

Notes

If your application uses delays or timeouts, `OSTimer()` must be called at the desired system tick rate. This is typically every 10-100ms. If your processor has limited resources, it may be unacceptable to dedicate a (relatively slow) timer resource to `OSTimer()`. By using `OSTIMER_PRESCALAR` you can call `OSTimer()` at one rate but have it actually perform its timer-related duties at a much slower rate, as dictated by the value of `OSTIMER_PRESCALAR`.

Unlike some hardware prescalars, which provide powers-of-2 prescaling (e.g. 1:2, 1:4, ...), the Salvo timer prescalar is implemented with a simple countdown timer, and can therefore provide a prescalar rate anywhere from 1:2 to 1:(2^32)-1.

A prescalar value of 1 accomplishes nothing and should not be used.

Whenever `OSTimer()` is called and its prescalar has not reached 0, a minimum of housekeeping is performed. When the prescalar reaches zero, `OSTimer()` increments the system tick count (if enabled), and the scheduler processes delayed and/or timed-out tasks.

OSTYPE_TCBEXT0|1|2|3|4|5: Set Tcb Extension Type

Name:	OSTYPE_TCBEXT0|1|2|3|4|5
Purpose:	To allow you to change the type of a tcb extension.
Allowed Values:	Any valid C-language type.
Default Value:	void *
Action:	Redefines OStypeTcbExt0|1|2|3|4|5.
Contained in:	salvo.h
Related:	OSENABLE_TCBEXT0|1|2|3|4|5, OScTcbExt0|1|2|3|4|5, OStcbExt0|1|2|3|4|5
Enables:	–
Memory Required:	Dependent on definition – affects size of tcbs.

Notes

A tcb extension can be of any valid type, and can have memory type qualifiers applied to it so long as they do not conflict with existing OSLOC_XYZ configuration options.

To use tcb extensions, the associated OSENABLE_TCBEXT0|1|2|3|4|5 must be set to TRUE.

See the example for OSENABLE_TCBEXT0|1|2|3|4|5 for more information.

OSUSE_CHAR_SIZED_BITFIELDS: Pack Bitfields into Chars

Name:	OSUSE_CHAR_SIZED_BITFIELDS
Purpose:	To reduce the size of Salvo objects.
Allowed Values:	FALSE: Places Salvo bitfields into int-sized objects. TRUE: Places Salvo bitfields into char-sized objects.
Default Value:	FALSE
Action:	Alters the typedef for OStypeBitField.
Contained in:	salvo.h, portxyz.h
Related:	–
Enables:	–
Memory Required:	When FALSE, reduces RAM requirements slightly.

Notes

ANSI C supports bitfields in structures. Multiple bits are combined into a single int-sized value, e.g.:

```
typedef struct {
  int field0:2;
  int field1:1;
  int field2:4;
  } bitfieldStruct;
```

Some compilers (e.g. HI-TECH PICC, Keil C51) allow the packing of bitfields into a single char-sized value in order to save memory. To use this feature, set OSUSE_CHAR_SIZED_BITFIELDS to TRUE. The Salvo type OStypeBitField will be of type char.

Not all compilers support this feature. If you are having problems compiling a Salvo application, set OSUSE_CHAR_SIZED_BITFIELDS to FALSE. The Salvo type OStypeBitField will then be of type int.

OSUSE_EVENT_TYPES: Check for Event Types at Runtime

Name:	`OSUSE_EVENT_TYPES`
Purpose:	To check for correct usage of an ecb pointer.
Allowed Values:	`FALSE`: Event-type error checking is not performed. `TRUE`: When using an event service (e.g. `OSSignalSem()`), Salvo verifies that the event being operated on is correct for the service.
Default Value:	`TRUE`
Action:	If `TRUE`, enables code to verify that the event type is what the service expects. This requires additional ROM, and a byte is added to each ecb (RAM).
Contained in:	`salvo.h`, `binsem.c`, `event.c`, `msg.c`, `msgq.c`, `rpt.c`, `sem.c`
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires a moderate amount of ROM.

Notes

Salvo uses event control block (ecb) pointers as handles to events. These pointers are passed as arguments to user event services (e.g. `OS_WaitMsg()`). A user might inadvertently pass an ecb pointer for one type of event (e.g. a semaphore) to a service for another type of event (e.g. `OSSignalMsg()`). The result would be unpredictable. Therefore an extra layer of error checking can be enabled to ensure that your application is protected against this sort of error.

Caution If you disable this configuration option you must be especially careful with event service arguments. The use of `#define` statements with descriptive names (e.g. `SEM1_P`, `SEM_COM1_P`, `MSG12_P`) for ecb pointers is highly recommended.

OSUSE_INLINE_OSSCHED: Reduce Task Call…Return Stack Depth

Name:	`OSUSE_INSELIG_MACRO`
Purpose:	To reduce the call…return stack depth at which Salvo tasks run.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, `OSSched()` is called as a function, and Salvo tasks run at a call…return stack depth of 1 greater than that of `OSSched()`. If `TRUE`, `OSSched()` is used in an inline form (i.e. macro), which reduces its call…return stack depth by 1.
Contained in:	`salvo.h`, `sched.c`
Related:	`OSUSE_INLINE_OSTIMER`
Enables:	–
Memory Required:	When `FALSE`, a small amount of extra ROM and one additional call…return stack level are used by `OSSched()`. When `TRUE`, `OSSched()` uses less ROM and only one call…return stack level.

Notes

Normally, you will call Salvo's scheduler in your application like this:

```
main()
{
  …
  OSInit();
  …
  for (;;)
    OSSched();
}
```

Since `OSSched()` calls Salvo tasks indirectly via function pointers, each task will run with two return addresses pushed onto the target processor's call…return stack: one inside of `OSSched()`, and one inside of `main()`.[63] This means that the call…return stack depth available to your functions called from within a Salvo task is equal to 2 less than the target processor's maximum call…return stack depth.

[63] This assumes that the compiler uses a `goto main()`, and calls all functions inside of `main()` from a call…return stack level of 0. Also, interrupts would add additional return addresses to the call…return stack.

If your target processor's call…return stack depth is limited, and you make deep, nested calls from within Salvo tasks or interrupt routines, you may want to reduce the call…return stack depth at which Salvo tasks run. By setting `OSUSE_INLINE_OSSCHED` to `TRUE`, and calling the scheduler like this:

```
main()
{
  …
  OSInit();
  …
  for (;;) {
    #include "sched.c"
  }
}
```

you can make Salvo tasks run with one fewer return addresses on the call…return stack, thereby freeing up one call…return stack level for other functions.

OSUSE_INLINE_OSTIMER: Eliminate OSTimer() Call…Return Stack Usage

Name:	`OSUSE_INLINE_OSTIMER`
Purpose:	To enhance ISR performance and reduce Salvo's call…return stack usage.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, OSTimer() is called as a function from an ISR. If `TRUE`, uses a macro to perform the same operation.
Contained in:	`salvo.h`, `timer.c`
Related:	`OSUSE_INLINE_OSTIMER`
Enables:	–
Memory Required:	When `FALSE`, a small amount of extra ROM and one call…return stack level are used by `OSTimer()`. When `TRUE`, `OSTimer()` uses less ROM and no call…return stack levels.

Notes

Normally you might call `OSTimer()` like this from your Salvo application:

```
void interrupt PeriodicIntVector ( void )
{
  …
  OSTimer();
}
```

This works for many applications. However, there may be disadvantages that arise when calling `OSTimer()` from an ISR. They include slower interrupt response time and larger code size due to the overhead of a call…return chain of instructions through `OSTimer()`and the need to save context during interrupts, and the consumption of one call…return stack level.

You can avoid all of these problems by setting `OSUSE_INLINE_OSTIMER` to `TRUE` and using `OSTimer()` like this:

```
void interrupt PeriodicIntVector ( void )
{
  …
  { #include "timer.c" }
}
```

This will insert an in-line version of `OSTimer()` into your ISR.

OSUSE_INSELIG_MACRO: Reduce Salvo's Call Depth

Name:	`OSUSE_INSELIG_MACRO`
Purpose:	To reduce Salvo's maximum call depth and parameter RAM usage.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`TRUE`
Action:	If `FALSE`, uses a function to perform a common operation internal to Salvo. If `TRUE`, uses a macro to perform the same operation.
Contained in:	`salvo.h`, `qins.c`
Related:	–
Enables:	–
Memory Required:	When `FALSE`, requires a small amount of ROM and may require extra RAM on the stack. When `TRUE`, requires a moderate amount of ROM.

Notes

If your processor is severely RAM-limited, you should leave this configuration option at its default value. For those processors that have a lot of RAM available (e.g. those with a general-purpose stack), then by setting `OSUSE_INSELIG_MACRO` to `FALSE` you should realize a reduction in code size at the expense of an additional call level and the RAM required to pass a tcb pointer as a parameter.

OSUSE_MEMSET: Use memset() (if available)

Name:	OSUSE_MEMSET
Purpose:	To take advantage of the presence of a working memset() library function.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, your code will use Salvo functions to clear global Salvo variables. If TRUE, memset() will be used to clear global Salvo variables.
Contained in:	portXyz.h, salvo.h, util.c
Related:	OSLOC_XYZ
Enables:	–
Memory Required:	Requires some ROM when FALSE.

Notes

Compilers will often use the standard library function memset() to clear (zero) global variables in start-up code.

If your target processor has a linear organization for RAM, you should probably set OSUSE_MEMSET to TRUE.

If you target processor uses banked memory, memset() may not work correctly for certain settings of OSLOC_ECB and OSLOC_TCB. In these cases, you should set OSUSE_MEMSET to FALSE in order to use Salvo's explicit byte-by-byte structure clearing functions.

Other Symbols

The following symbols are used in the Salvo distribution. They are not part of Salvo per se, and therefore do not carry the OS prefix.

MAKE_WITH_FREE_LIB, MAKE_WITH_STD_LIB: Use salvocfg.h for Multiple Projects

Name:	MAKE_WITH_FREE_LIB, MAKE_WITH_STD_LIB
Purpose:	To enable a single salvocfg.h to serve more than one project.
Allowed Values:	undefined or defined
Default Value:	undefined
Action:	If defined, can be used to exclusively define symbols for use with the freeware libraries.
Contained in:	The salvocfg.h of various projects in the Salvo distribution.
Related:	OSLIBRARY_CONFIG, OSLIBRARY_TYPE, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Enables:	–
Memory Required:	n/a.

Notes

Certain projects in the Salvo distribution are made with the freeware and/or standard libraries. In order to simplify the directory / folder structures, a single salvocfg.h configuration file is used for the same application built from either the Salvo source code or the freeware or standard libraries.

Each library-based project in the Salvo distribution is compiled with the MAKE_WITH_XYZ_LIB symbol defined, usually via one of the compiler's command-line options.[64] Below is an example[65] of a salvocfg.h file that uses MAKE_WITH_FREE_LIB and MAKE_WITH_STD_LIB:

```
#if !defined (MAKE_WITH_FREE_LIB) && !defined
(MAKE_WITH_STD_LIB)

#define OSCOMPILER              OSHT_PICC
#define OSEVENTS                0
#define OSTARGET                OSPIC16
```

64 E.g. -Dsymbol for the HI-TECH PICC compiler.

65 tutorial\tu1\sysa.

```
#define OSTASKS                 2
#define OSLOC_ALL               bank1

#else

#define OSUSE_LIBRARY           TRUE
#ifdef MAKE_WITH_FREE_LIB
#define OSLIBRARY_TYPE          OSF
#elif defined(MAKE_WITH_STD_LIB)
#define OSLIBRARY_TYPE          OSL
#endif
#define OSLIBRARY_CONFIG        OSM
#define OSLIBRARY_VARIANT       OSNONE

#endif
```

Listing 32: salvocfg.h for Multiple Projects

The `#ifndef` … `#else` … `#endif` preprocessor directives above will result in the first group of configuration options being used when the project is built from Salvo source files and neither `MAKE_WITH_FREE_LIB` nor `MAKE_WITH_STD_LIB` is defined. The second group will be used if/when either `MAKE_WITH_FREE_LIB` or `MAKE_WITH_STD_LIB` is defined and will facilitate linking to the appropriate freeware library.

By controlling which part(s) of `salvocfg.h` are used for a particular build, multiple project files[66] can exist in the same directory along with a single `salvocfg.h`.

See *Chapter 8 • Libraries* for more information on using libraries.

66 E.g. Microchip MPLAB's `.pjt` project files.

SYSA|B|…|Z: Identify Salvo Test System

Name:	SYSA|B|…|Z
Purpose:	To identify Salvo test system hardware for proper hardware configuration in a particular main.c.
Allowed Values:	undefined or defined. Only one test system should be defined at any time.
Default Value:	undefined
Action:	If defined, can be used in main.c to configure source code for a particular test system.
Contained in:	The project file and / or salvocfg.h of various projects in the Salvo distribution.
Related:	–
Enables:	–
Memory Required:	n/a.

Notes

Many projects in the Salvo distribution are designed to run on different test systems. It often is the case that certain objects (e.g. LEDs, switches, analog inputs, A/D converter registers) vary from test system to test system. SYSA through SYSZ are used in salvocfg.h to identify the test system in use for the project. This allows a single main.c to function as the source code for several different projects.

```
#if defined(SYSF)

__CONFIG(1, FOSC0 | UNPROTECT);
#define LED_PORT     PORTB
#define LED_TRIS     TRISB
#define ADGO_BIT     GODONE
#define ADREG        ADRESH
static bit keySW @ PORTBIT(PORTA, 4);

#elif defined(SYSH)

__CONFIG(FOSC0 | UNPROTECT);
#define LED_PORT     PORTC
#define LED_TRIS     TRISC
#define ADGO_BIT     ADGO
#define ADREG        ADRESH
static bit keySW @ PORTBIT(PORTB, 0);

#endif
```

Listing 33: Use of SYSA … SYSZ in main.c

In Listing 33 the upper group of configuration option, symbol definitions and variable declaration is used with a Microchip PIC18C452 microcontroller running on a Microchip PICDEM-2 demonstration board. The lower group is used when running the same application on a Microchip PIC16F877 with a Microchip MPLAB-ICD. The PICDEM-2's LEDs are on I/O port B, whereas the MPLAB-ICD's are on I/O port C. Similarly, the 18C452's A/D converter's Go/Done bit is defined as `GODONE` in the compiler's header file, whereas the PIC16F877's is defined as `ADGO`.

The `salvocfg.h` for Salvo Test System F is shown in Listing 34.

```
#define SYSF                        TRUE

#if defined MAKE_WITH_FREE_LIB

#define OSUSE_LIBRARY               TRUE
#define OSLIBRARY_TYPE              OSF
#define OSLIBRARY_CONFIG            OSA
#define OSLIBRARY_VARIANT           OSB

#endif
```

Listing 34: Use of SYSA … SYSZ in salvocfg.h

See *Appendix C • File and Program Descriptions* for more information on Salvo's test systems.

USE_INTERRUPTS: Enable Interrupt Code

Name:	`USE_INTERRUPTS`
Purpose:	To control compilation of interrupt code in certain Salvo projects.
Allowed Values:	undefined or defined.
Default Value:	undefined
Action:	If defined, is used in `isr.c` and / or `isr.h` to configure interrupt code for a particular test system.
Contained in:	The project file and / or `salvocfg.h` of various projects in the Salvo distribution.
Related:	–
Enables:	–
Memory Required:	n/a.

Notes

Many projects in the Salvo distribution are designed to run on different test systems. Interrupt code often varies from test system to test system. Where interrupt code is required, `USE_INTERRUPTS` is used to enable it. This allows a single `isr.c` to function as the interrupt source code for several different projects.

```
#include "isr.h"
#include <salvo.h>

#if defined(USE_INTERRUPTS)

#if defined(SYSA) || defined(SYSH) || de-
fined(SYSF)

void interrupt IntVector( void )
{
    if ( T0IE && T0IF ) {
        T0IF = 0;
        TMR0 -= TMR0_RELOAD;

        OSTimer();
    }
}


#elif defined(SYSI)

void timer0 ( void) interrupt 1 using 2
{
    OSTimer();
}

…

#endif /* defined(SYSA) || … */

#endif /* defined(USE_INTERRUPTS) */
```

Listing 35: Use of USE_INTERRUPTS in isr.c

Obsolete Configuration Parameters

The following configuration parameters are obsolete and no longer supported. Including them in your `salvocfg.h` will result in a compile-time error message. Some error messages include instructions on alternate, renamed or related configuration options.

As of 3.1.0

```
OSCALL_OSCREATEBINSEM
OSCALL_OSCREATEMSG
OSCALL_OSCREATEMSGQ
OSCALL_OSCREATESEM
OSCALL_OSSIGNALBINSEM
OSCALL_OSSIGNALMSG
OSCALL_OSSIGNALMSGQ
OSCALL_OSSIGNALSEM
OSPIC16_GIE_BUG
OSSUPERTIMER_PRESCALAR
OSTEST_SYSTEM_A|B|…|Z
OSUSE_CIRCULAR_QUEUES
OSUSE_INSELIGQ_MACRO
OSUSE_SUPERTIMER
```

Listing 36: Obsolete Configuration Parameters

Chapter 6 • Frequently Asked Questions (FAQ)

General

What is Salvo?

Salvo is a powerful and feature-rich real-time operating system (RTOS) for single-chip microcontrollers with limited ROM and RAM.

Salvo is so small that it runs where other RTOSes can't. Its RAM requirements are minuscule, and it doesn't need much ROM, either.

Salvo is not a state machine. It is not a "a neat trick." It is not an app note. Salvo is all the RTOS code you need and more to create a high-performance embedded multitasking program in systems where kilobytes of ROM are a luxury and available RAM is measured in tens of bytes.

Is there a shareware / freeware / open source version of Salvo?

There is a freeware version called Salvo Lite.

Processor- and compiler-specific freeware libraries are provided as part of each Salvo Lite distribution. *Each freeware library supports a limited number of tasks and events.* All of the default functionality is included in the freeware libraries. If you need more tasks and/or events, or you need access to Salvo's advanced functionality, then you should consider purchasing the full version of Salvo.

The full version of Salvo includes all source code. Source code is not included[67] in Salvo Lite. Salvo is not open source.

---

67 Except for a few specific files in certain freeware versions.

Just how small is Salvo?

On a RISC-based single-chip microcontroller, a typical[68] multitasking application might need a several hundred instructions in ROM and around fifty bytes of RAM for all of Salvo's code and data.

Why should I use Salvo?

If you want to:

- get your embedded product to market ahead of the competition,
- add greater software functionality to your existing hardware design,
- improve the real-time performance of a complex design,
- not have to re-invent the wheel,
- have a powerful framework to do multitasking programming,
- control the increasing complexity of your applications,
- minimize your hardware costs by using smaller and cheaper processors,
- not be left behind by the multitasking / RTOS wave and/or
- maximize the reliability of your complex applications

then Salvo is for you.

Low-cost single-chip microcontrollers are capable of hosting sophisticated real-time applications, but programming them to do so can be quite a challenge. Real-time kernels can simplify the design of complex software. They provide proven mechanisms to accomplish a variety of well-understood operations within predictable time frames. Unfortunately, most commercial real-time offerings require large amounts of ROM and RAM – requirements that are largely incompatible with these chips. Programmers of low-end embedded processors have been at a disadvantage when developing non-trivial applications.

Salvo changes all of that. Now you can develop applications for inexpensive one-chip microcontrollers similar to how you would for a Pentium® in an embedded application.

---

68 Microchip® PIC16C64 with five concurrent tasks and five events.

Salvo will get your application up and running quickly. It provides you with a clean and easily-understood multitasking framework that uses a minimum of memory to get the job done.

What can I do with Salvo?

You can throw out any preconceived notions on how difficult or time-consuming embedded programming can be. You can stop dreaming about multiple, independent processes running concurrently in your application without crashing. You can reorganize your code and no longer worry about how a change in one area might affect another. You can add new functionality to your existing programs and know that it will integrate seamlessly. You can easily link external and internal events to program action.

Once you start creating applications with Salvo, you can focus on adding functionality to and improving the performance of your application by creating tasks and events tailored specifically to it. You can create multitasking applications where tasks pass information to other tasks and the rest of your application. You can prioritize the tasks so that your processor is spending its time doing what's most important, instead of unnecessary housekeeping chores. You can have events control how and when tasks run. You can worry a lot less about interrupts. You can write powerful, efficient and reliable multitasking applications with predictable real-time performance.

And you can do all of this a lot more quickly than you'd expect.

What kind of RTOS is Salvo?

Salvo is a priority-based, event-driven, cooperative, multitasking RTOS. It is designed to run on processors with severely limited resources (primarily ROM and RAM).

What are Salvo's minimum requirements?

Salvo requires a full-featured ANSI-C-compliant C compiler from a third party. Contact the factory for a list of tested and/or approved compilers. The Salvo files require less than 30MB of hard disk space.

If you're not already reasonably proficient in C, you will need to review certain concepts (particularly pointers, if you plan on using messages and message queues) before beginning with Salvo. You don't need to be an expert C programmer to use Salvo.

What kind of processors can Salvo applications run on?

Salvo requires a processor with a hardware call…return stack of at least 4 levels and enough memory for Salvo's code and data. ROM and RAM requirements vary, and are controlled primarily by your application's source code and settings in the Salvo configuration file `salvocfg.h`.

Salvo does not require, nor does it use, a general-purpose stack.[69] It can run on stack-less processors as well as any processor with a stack, from a PICmicro® to a Pentium®.

How many tasks and events does Salvo support?

Salvo supports an unlimited number of tasks and events. The number of tasks and events in your application is limited only by available RAM. Salvo's default configuration supports up to 255 tasks, 255 events and 255 message queues.

How many priority levels does Salvo support?

Salvo supports 16 distinct priority levels. Tasks can share priority levels.

What kind of events does Salvo support?

Salvo supports binary semaphores, counting semaphores, event flags, messages and message queues. You can create ("init") events, signal ("post", "put", "unlock", "release", "send") events and have tasks wait ("pend", "get", "lock", "acquire", "receive") on each event.

[69] A stack pointer (SP) and/or PUSH and POP instructions are evidence of a general-purpose stack.

Is Salvo Y2K compliant?

Yes. Salvo does not provide any functions for reporting or setting the absolute time of day and date (e.g. 10:22.36pm, Nov. 11, 1999). Therefore Salvo is by definition Y2K compliant.

Where did Salvo come from?

Salvo 1.0 was originally developed in assembly language for use in a low-cost, high-performance multichannel racecar data acquisition system. Its appeal to a wider audience was quickly recognized, whereupon it was rewritten in C for greater portability and configurability.

Getting Started

Where can I find examples of projects that use Salvo?

Salvo distributions include `demo`, `tutorial`, `example` and `test` folders. Refer to *File and Program Descriptions* in the *Salvo User Manual* for a test system (e.g. `sysa`) that's similar to yours. Then search these folders in your Salvo installation for project files, source code (usually `main.c`) and configuration files (`salvocfg.h`).

Is there a tutorial?

Yes. An in-depth tutorial can be found in the *Salvo User Manual*.

Apart from the Salvo User Manual, what other sources of documentation are available?

The *Application Notes* contain information on a verity of topics.

I'm on a tight budget. Can I use Salvo?

You can use the freeware version of Salvo, with its complete set of freeware libraries, to create fully functioning Salvo applications. You'll be limited to the numbers of tasks and events your application can support.

I only have an assembler. Can I use Salvo?

No. You will need a certified C compiler to use Salvo.

Performance

How can using Salvo improve the performance of my application?

If you're used to programming within the conventional foreground / background loop model, converting your application to a Salvo application may yield substantial performance benefits.

For example, it's not uncommon to write a program that polls something (say an I/O pin) repeatedly and performs a complicated and time-consuming action whenever the pin changes. You might have a timer interrupt which calls a subroutine to poll a port pin and XOR it against its previous value. If the pin changes, then you might set a bit in a global status byte, which is then tested every time through your main loop. If the bit is set, you disable interrupts, clear the status bit, reenable interrupts and then take an appropriate action.

The problem with this approach is that your program is consuming processor cycles while sampling information that remains unchanged for most of the time. The more infrequently the event (in this case, the change on I/O pin) occurs, the more inefficient your program is.

The solution is to employ an event-based approach by using Salvo. When a task is made to wait an event, and the event is not available (e.g. the I/O pin hasn't changed), then the task is put into a waiting state. From this time forward, until the event occurs, not a single processor cycle is expended on waiting for the event. Zip, zero, nada. When the event does finally occur, the task will process the event as soon as it is made to run by the scheduler. In other words, it's the event that drives all the other actions directly. With events driving your application, it can spend its time on the most important things, as defined by you, the programmer.

It's important that you understand the distinction between polled and event-based actions.

How do delays work under Salvo?

Salvo provides a simple means of delaying tasks. While a task is delayed, it consumes a minimum of processor resources, and your other (non-delayed) tasks can continue to run. The overhead to support one or more delayed tasks is the same. You can specify delays to the resolution of the system timer, which is under your control.

See the *Timer and Timing* section in this FAQ for more information.

What's so great about having task priorities?

The point of assigning priorities to tasks is to make the most of your processor's power by having it always doing what is most important at that particular instant in time.

For example, say you have an instrument whose primary purpose is to generate moderate-frequency waveforms. But you'd also like to monitor various analog voltages in the instrument to ensure no out-of-range conditions. By assigning the waveform-generating task a high priority, and the analog-sampling task a low priority, the Salvo application will automatically run the sampling task when there's no demand for the waveform to be generated. But while the waveform is being generated, the sampling task will not interfere.

All you have to do in Salvo is assign each task an appropriate priority, and ensure that each task context-switches often enough to allow other tasks to run as needed.

When does the Salvo code in my application actually run?

Salvo's code runs only when you explicitly call Salvo's user services within your application. In most cases it's pretty obvious when your processor is running Salvo code – for example, when you start a task by calling `OSCreateTask()` or `OSStartTask()`.

When the scheduler and timer actually run is perhaps a little less obvious. The scheduler runs as part of any context switch in your code, and it also runs when there are no tasks eligible to run. The timer runs whenever it is called at the periodic system timer rate, which is usually done via a periodic interrupt.

How can I perform fast, timing-critical operations under Salvo?

In order to control critical timing under any RTOS, follow these two rules: 1) give timing-critical tasks high priorities, and 2) use Salvo's flexible features to prevent or delay it from doing anything during a critical time period.

Since Salvo is a cooperative multitasking RTOS, during a timing-critical task there is only one source of potential interference – interrupts. Interrupts which might involve Salvo would be those that signal events and / or call the system timer `OSTimer()`. By preventing calls to Salvo services during timing-critical operations you can guarantee the proper operation of your system.

If, on the other hand, your application can tolerate the timing jitter that will occur if Salvo services are invoked during a critical period, then you may not have much to worry about. This is usually the case with operations whose frequency is much less (e.g. 1/50) than that of the system timer.

Memory

How much will Salvo add to my application's ROM and RAM usage?

Salvo's ROM requirements depend on how many of its functions you call, and its RAM requirements depend on how many tasks and resources you create. Salvo was specifically designed for processors with limited memory resources, and so it requires only a small fraction of what a typical multitasking kernel would normally need.

The Salvo User's Manual contains specific information on memory requirements for a variety of representative test systems.

How much RAM will an application built with the libraries use?

Using a PIC16 library[70] that supports multitasking, delays, and events (binary and counting semaphores, as well as messages), an application will need

- 10 bytes of RAM for Salvo's global variables[71]
- 5 bytes of RAM per task
- 3 bytes of RAM event

The compiler will need some additional RAM to handle local variables, interrupt save and restore, etc. But the numbers above represent how little RAM Salvo needs to implement all its functionality.

Do I need to worry about running out of memory?

No. Salvo's RAM memory requirements are fixed at compile time. They are simply:

```
  #(tasks) x sizeof(task control block)
+ #(events) x sizeof(event control block)
+ #(tcb pointers[72]) x sizeof(tcb pointer)
+ #(message queues) x sizeof(message queue control
  block)
+ #(message queues) x sizeof(user-defined message
  queues)
+ sizeof(variables associated with configuration
  options)
```

These requirements do not change during runtime, and are not dependent on call depth, the status of any of the tasks, the values of any of the events or any other multitasking-related issues. Once you define tasks and events in Salvo and your application has the memory to support them, you can do whatever you want without the fear of running out of memory.

Salvo cannot "run out of memory" during runtime.

---

70 `sfp42Cab.lib`, for the PIC16F877 for use with the HI-TECH PICC compiler.

71 4 of the 10 bytes of global variables are for the 32-bit elapsed time counter, which can be disabled by doing a source-code build (no libraries).

72 2 or 3, depending on the configuration.

If I define a task or event but never use it, is it costing me RAM?

Yes. The RAM memory is allocated at compile time.

How much call ... return stack depth does Salvo use?

Normal stack depth is 4, and in some instances Salvo can be configured to use a maximum call…return stack depth of 3. This means that no Salvo function will require a call-return stack more than 4 levels deep, not including interrupts. This is accomplished by setting the following configuration parameters in your `salvocfg.h`:

```
#define OSLOGGING                FALSE
#define OSUSE_INLINE_OSSCHED     TRUE
#define OSUSE_INLINE_OSTIMER     TRUE
#define OSUSE_OSINSELIGQ_MACRO   TRUE
```

and making the appropriate changes to your source code (see the configuration options' descriptions for more information). These options will configure Salvo to use in-line forms of various functions (thus saving one or more call…return stack levels) and to use simple function return codes without debug messages (saving another call…return stack level).

When calling Salvo functions (e.g. `OSSignalMsg()`) from ISRs, remember that ISRs are likely to run one or more stack levels deep, depending on when the interrupt is serviced. This will affect the maximum call ... return stack depth in your application.

By choosing `OSENABLE_STACK_CHECKING` Salvo will monitor the stack depth of all of its functions and report back the maximum stack depth reached. This is especially useful when simulating your application by running Salvo on a PC.

Note that the numbers above are based on Salvo's inherent call...return tree, and do not include any additional stack depth due to how your compiler does certain things like indirect function calls.

Why must I use pointers when working with tasks? Why can't I use explicit task IDs?

Salvo user services originally took task, event and message queue IDs (simple integer constants) as parameters to refer to Salvo ob-

jects. The advantage of this approach was that it was very easy for beginners to understand, it easily accommodated run-time error checking, and the memory requirements (mainly when passing parameters) were minimal. However, it also had several severe disadvantages, including increased code size, lack of flexibility, poor run-time performance and increased call…return stack usage.

Salvo services now use pointers as parameters to refer to Salvo objects. Along with the attendant advantages that pointers bring with them, Salvo's syntax is more like other, larger RTOSes. Somewhat surprisingly, the memory requirements actually *decreased* for many target processors.

With the pointer-based approach, the simplest way to refer to a task is to use the `OSTCBP()` macro, which returns a pointer to the tcb of a particular task. This is a compile-time constant (it's an address of an array element), and on many targets[73] uses the same amount of memory as an 8-bit integer constant. Similar macros exist for events, message queues, etc. These macros allow you to refer to Salvo objects explicitly.

An alternative approach is to use a *handle*, a variable that contains a pointer to a particular task's tcb. This offers flexibility but has the disadvantage that it consumes extra RAM. For some applications handles can be very useful.

Using the C `#define` preprocessor directive for event IDs can substantially improve code legibility. For example, use:

```
/* pointer to display binSem. */
#define  BINSEM_DISP_P  OSECBP(3)

/* create display semaphore, init to 1. */
OSCreateSem(BINSEM_DISP_P, 1);
...
/* get display. */
OS_WaitSem(BINSEM_DISP_P, OSNO_TIMEOUT, label);
...
/* release display. */
OSSignalSem(BINSEM_DISP_P);
```

to reference the binary semaphore that is used as a resource to control access to a display in a easy-to-read manner.

[73] E.g. PIC16 and PIC17 series of PICmicro MCUs.

How can I avoid re-initializing Salvo's variables when I wake up from sleep on a PIC12C509 PICmicro MCU?

The PIC12C509 has a simple architecture (no interrupts, single reset vector) and always vectors to the last location in ROM when it wakes from sleep due to the watchdog timer or wake-on-pin-change. Normally, the startup code generated by the compiler will initialize all static and global variables immediately after any type of reset – power-on reset (POR) or otherwise. This will reset all of Salvo's variables to 0, equivalent to calling `OSInit()`.

Since you'd like to preserve the state of your multitasking system on wake-from-sleep, and not reset it, you must declare Salvo's variables to be of type persistent. This instructs the compiler to skip the initialization for these variables. If you are using HI-TECH PICC, the easiest way to declare Salvo's variables as persistent is to use the `OSLOC_ALL` configuration option, like this:

```
#define OSLOC_ALL bank1 persistent
```

This will place all of Salvo's variables in RAM bank 1, and will prevent the startup code (which is executed after every type of reset, not just POR) from resetting the variables to zero. If you use this method, you *must* call `OSInit()` after each POR (and not after other types of reset) in order to properly initialize Salvo.

Libraries

What kinds of libraries does Salvo include?

Every Salvo distribution includes the freeware Salvo libraries. Additionally, the full version includes the standard Salvo libraries. There are many different library types, depending on how much functionality you need.

What's in each Salvo library?

Each Salvo library contains the default Salvo functionality for the particular library type. Additionally, each library is compiled for a default number of Salvo objects (tasks, events, etc.). Some libraries (notably those for targets with extremely limited RAM) have a subset of the normal functionality.

Why are there so many libraries?

Each library is generated with a particular compiler, target processor and library type in mind. As a result, a large number of libraries is required to span all the possible combinations.

Should I use the libraries or the source code when building my application?

If you don't have the full version, you'll have to use the freeware libraries.

With the full version, you should use the standard libraries until you reach a situation where the configuration of the library no longer suits your application, e.g. you want 32-bit delays and the library supports only 8-bit delays. In that case, you'll need to use the source code and some configuration options to achieve the desired Salvo functionality. You can always generate your own Salvo libraries, too.

What's the difference between the freeware and standard Salvo libraries?

There is very little difference. The freeware libraries are limited to a maximum number of Salvo objects. The standard libraries support as many Salvo objects as you can fit in RAM. Both library types are compiled for the same number of objects.

My library-based application is using more RAM than I can account for. Why?

The default number of Salvo objects used by each library requires a certain amount of RAM, whether or not you use all of those objects. If your application uses fewer objects, you can reduce the application's RAM requirements by compiling and linking the `mem.c` module with a different set of configuration objects. See *Chapter 8 • Libraries* for more information.

I'm using a library. Why does my application use more RAM than one compiled directly from source files?

Each library is created with its own default configuration. Some configurations include Salvo features that require one or more

bytes of RAM. For example, the library may be configured to support a single message queue as well as other event types. Each message queues requires its own message queue control block (mqcb), and RAM has been allocated for it in the library. Therefore even if you do not use message queues in your application when linking to a library, RAM is allocated for this (unused) message queue.

You can reduce some of the library's RAM requirements by overriding the RAM allocations. See *Chapter 8 • Libraries* for more information.

I'm using a freeware library and I get the message "#error: OSXYZ exceeds library limit – aborting." Why?

You've probably set `OSXYZ` to a number that exceeds the maximum value supported by the library. Remove `OSXYZ` from your `salvocfg.h`.

I'm using a standard library and I can't increase the number of tasks beyond the library's default. Why?

If your application needs more tasks than the standard library supports, you can increase the RAM allocated to Salvo's variables by compiling and linking the `mem.c` module with a different set of configuration objects. See *Chapter 8 • Libraries* for more information.

Why can't I alter the functionality of a library by adding configuration options to my salvocfg.h?

The configuration options affect a library only at compile time. Since the libraries are precompiled, changing configuration options in your `salvocfg.h` will have no effect on them. Choose a different library with the functionality you desire, or use the source code.

The libraries are very large – much larger than the ROM size of my target processor. Won't that affect my application?

No. Your compiler will extract only the modules that it needs from the library you're using. In fact, linking to libraries creates the smallest possible Salvo applications.

Why is there a precompiled mem.c object in each library?

By having the `mem.c` (*memory*, i.e. Salvo's variables) object in each library, building an application is as simple as linking your source code to a single library. Without it, you'd have to add `mem.c` to your project, and set additional configuration parameters in order to build the application. That's also how you can override the memory settings of the standard libraries.

I'm using a library. Can I change the bank where Salvo variables are located?

No. On banked target processors, the locations of the Salvo variables are determined by the library. To "move" the variables to another bank, you'll need to use the source files, set your own configuration options, and recompile.

Configuration

I'm overwhelmed by all the configuration options. Where should I start?

If you're using a Salvo library, the only configuration options you need are the ones that tell Salvo which kind of library you're using. You needn't worry too much about the others.

If you have the full version, or you want more objects than are supported by default in the standard libraries, you'll find various configuration options useful when tailoring Salvo to your application. Start with the default configuration, which you'll find in `salvo\inc\user\salvocfg.h` . Copy it to your own `salvocfg.h`. Specify your compiler and processor by adding the `OSCOMPILER` and `OSTARGET #defines`. Try to get your program

compiled successfully. Modify your `salvocfg.h` as you enable Salvo functionality that differs from the default.

Alternately, you can start with a blank `salvocfg.h` and add only those configuration options (e.g. `OSCOMPILER`, `OSTARGET` and `OSTASKS`) that need values other than the Salvo defaults. This is the preferred method.

Three good places to get acquainted with the configuration options and how they're used are the tutorial, test and demonstration programs in the standard Salvo distribution. By examining the programs and their corresponding `salvocfg.h` files you should be able to develop a feel for when to use a particular configuration option. These programs are found in `salvo\test` and `salvo\demo`.

Do I have to use all of Salvo's functionality?

You can use as little or as much as you like. Only those portions that you use will be incorporated into (i.e. will take up ROM and RAM in) your final executable. By choosing configuration options you can control how much functionality Salvo delivers to your application.

What file(s) do I include in my main.c?

For many target processors, including `salvo.h` is enough to automatically include the necessary processor-specific header files.

What is the purpose of OSENABLE_SEMAPHORES and similar configuration options?

Users who compile their applications by linking multiple Salvo source files may find this type of configuration option useful. That's because entire modules can be disabled simply setting the configuration option to `FALSE` in `salvocfg.h` instead of changing the setup to your compiler / project / IDE.

Can I collect run-time statistics with Salvo?

By enabling `OSGATHER_STATISTICS` Salvo will track and report the number of context switches, warnings, errors, timeouts and calls to the idle function (if enabled).

How can I clear my processor's watchdog timer with Salvo?

Good coding practice dictates that watchdog timers only be cleared from a single place within an application. An excellent place to do so is from within Salvo's scheduler. By defining `OSCLEAR_WATCHDOG_TIMER()` in `salvocfg.h` with the instruction(s) your application requires to clear the watchdog timer, Salvo will execute the instruction(s) each time the scheduler is called. Therefore, if a task fails to release control back to the scheduler, the watchdog will time out, indicating a fault.

I enabled timeouts and my RAM and ROM grew substantially– why?

Salvo makes the most efficient use of RAM and ROM based on the configuration options you've chosen. Adding support for timeouts requires an additional amount of RAM for each task, and extra code in ROM, in order to support a task's ability to wait on an event with a timeout. RAM- and ROM-wise, this is probably the most "expensive" Salvo configuration option.

Timer and Timing

Do I have to install the timer?

If you want to make any use of Salvo's time-based functions (task delays, timeouts when waiting for a resource, elapsed time, etc.) you must install the timer. Multitasking and support for events do not require the timer.

To use Salvo's delay and timeout features, configure `OSBYTES_OF_DELAYS` to a non-zero value appropriate for your application.

To use Salvo's elapsed time features, configure `OSBYTES_OF_TICKS` to a non-zero value appropriate for your application.

How do I install the timer?

In your application you must call `OSTimer()` at the tick rate you feel is appropriate for your application. Usually this is done by creating a periodic interrupt at the desired tick rate, and having the associated ISR call `OSTimer()`. `OSTimer()` must be called in only one place in your application.

I added the timer to my ISR and now my ISR is huge and slow. What should I do?

See "Why did my interrupt service routine grow and become slower when I added a call to `OSTimer()`" in this FAQ.

How do I pick a tick rate for Salvo?

The ideal Salvo "tick" rate is dependent on the application, and hence is configurable. Rates on the order of 10-100Hz are commonly used. The tick rate defines the timer resolution in Salvo, but does not directly affect the latency of a task made ready-to-run. The context-switching rate is independent of the tick rate. A faster tick rate requires more processor, but it gives better timer resolution, and may require additional memory for the delay fields in the task blocks.

Once you've chosen a tick rate, you must configure your system to call `OSTimer()` each time the tick occurs. This is usually done via a periodic interrupt.

How do I use the timer prescalar?

A linear prescalar for the Salvo timer is provided to create a slower Salvo "tick" rate independent of the timer to which the Salvo timer is chained. For example, on a 4MHz system with a hardware timer that generates interrupts at a 500 Hz rate (i.e. every 2 ms), by defining `OSTIMER_PRESCALAR` to 5 the desired Salvo tick rate will be 100Hz (i.e. every 10ms). The maximum value for the prescalar is (2^32)-1, and to disable it altogether simply set it to 0 (the default).

I enabled the prescalar and set it to 1 but it didn't make any difference. Why?

The Salvo timer prescalar is enabled if `OSTIMER_PRESCALAR` is set to a number greater than or equal to 1, resulting in prescalar rates of 1:1, 1:2, 1:3, ... 1:(2^32)-1. A prescalar value of 1 will add a few instructions to `OSTimer()` and will require a byte of RAM storage for `OStimerPS`, but it will not change the rate at which `OSTimer()` is called, since the prescalar rate is 1:1. In order to change the rate at which `OSTimer()` is called in your application, choose a value for the timer prescalar that is 2 or greater.

What is the accuracy of the system timer?

As long as the system tick rate is slow enough to give Salvo's system timer `OSTimer()` enough time to do its job, the system timer will have no more than 1 timer tick of inaccuracy.

What is Salvo's interrupt latency?

Salvo must disable interrupts while certain internal operations are being performed. Every effort has been made to minimize Salvo's interrupt latency. See the Salvo User's Manual for specifications on Salvo's interrupt latency.

What if I need to specify delays larger than 8 bits of ticks?

You have three options. You can change the configuration parameter `OSBYTES_OF_DELAYS` to use 16- or 32-bit delays instead of 8-bit delays. This will consume an additional 1 or 3 bytes of RAM per task, respectively.

You can call `OS_Delay()` multiple times (sequentially, or in a loop) to create longer delays.

Or you can make use of the `OSTIMER_PRESCALAR` configuration parameter. However, this approach will reduce the resolution of the system timer.

How can I achieve very long delays via Salvo? Can I do that and still keep task memory to a minimum?

The maximum delay and timeout length is user-configurable as (2^(n x 8))-1, where n is the size in bytes for the task's delay field. For example, if 16-bit delays are selected, delays and timeouts of up to 65535 clock ticks are possible. Since all tasks have the same-size delay field, the total amount of RAM memory dedicated to holding the delays is

sizeof(delay field) x #(tasks).

If your application uses delays and timeouts sparingly, but requires a very long timeout, you can use a small value for `OSBYTES_OF_DELAYS` (e.g. 1, for 1 byte / 8 bits / maximum count of 255) and nest the call within a local loop to achieve a multiple of the maximum timeout supported by Salvo. For example, using

```
for ( i = 0 ; i <= TIMEOUT_MULTIPLE ; i++ )
{
  OS_WaitSem(SEM_NAME_P, MAX_TIMEOUT, label);
  if ( !OSTimedOut() )
    break;
}
if ( OSTimedOut() ) {
  /* loop is over, are we here because of a    */
  /*  timeout or did we wait the semaphore     */
  /*  successfully?                            */
}
```

within a task (where the loop counter `i` is static) will result in a maximum timeout of `TIMEOUT_MULTIPLE x MAX_TIMEOUT`. With a looping construct like this a timeout or delay can be made arbitrarily long at the cost of only a single static variable local to the task of interest.

Note that many target processors do math efficiently only for their native data size. Therefore Salvo's timer code will grow substantially on an 8-bit PICmicro if you use 32-bit delays.

An alternative method is to use Salvo's timer prescalar. This method will affect all Salvo delays and timeouts, system-wide. In order to use Salvo's delays and timeouts `OSBYTES_OF_DELAYS` must be non-zero. In order to use the timer prescalar, `OSTIMER_PRESCALAR` must be set to a non-zero value.

Can I specify a timeout when waiting for an event?

Yes. When waiting for an event you can specify an optional timeout in system ticks. `OSENABLE_TIMEOUTS` must be `TRUE` in order to wait with timeouts.

Does Salvo provide functions to obtain elapsed time?

Yes. Salvo provides two elapsed time functions, `OSGetTicks()` and `OSSetTicks()`. These functions get and set, respectively, the current number of timer ticks since the free-running timer ticks counter rolled over. To use these elapsed time functions, the configuration parameter `OSBYTES_OF_TICKS` must be non-zero.

In this example, a task waits for a message, and once obtained, calculates the amount of elapsed time in timer ticks (`OSBYTES_OF_TICKS` is defined to be 4 in `salvocfg.h`):

```
...
static OStypeMsgP msgP;
static OStypeTick elapsedTicks;
...
for (;;) {
  ...
  OSSetTicks(0);
  OS_WaitMsg(MSG_ID, &msgP, OSNO_TIMEOUT, label);
  elapsedTicks = OSGetTicks();
  printf("%lu ticks have passed\n", elapsedTicks);
  ...
}
```

How do I choose the right value for OSBYTES_OF_TICKS?

Salvo uses a free-running counter to monitor system ticks. This counter is incremented by 1 each time the system timer `OSTimer()` is called by your application.[74] The size of this counter, and hence the rollover period, is controlled by the configuration parameter `OSBYTES_OF_TICKS`.

Since system ticks are used only for obtaining elapsed time and statistics, your choice for the value of `OSBYTES_OF_TICKS` is entirely dependent on the longest elapsed time you wish to be able to measure accurately.

---

74 For every `OSTIMER_PRESCALAR` calls to `OSTimer()` if `OSTIMER_PRESCALAR` is nonzero.

For example, let's assume that you have written your application to have an effective tick rate of 100Hz by enabling Salvo's system timer, choosing an appropriate value for `OSTIMER_PRESCALAR`, and calling `OSTimer()` from inside a timer-interrupt ISR. If `OSBYTES_OF_TICKS` were defined to be 2, the longest time interval you could measure would be (65535/100) seconds, or just under 11 minutes. If more than 11 minutes elapse before calling `OSGetTicks()`, the reported elapsed time will be the actual elapsed time modulo 11 minutes, an erroneous result.

My processor has no interrupts. Can I still use Salvo's timer services?

Yes. As long as you have some form of a timer, you can use `OSTimer()`. For example, you can monitor a free-running counter for overflow, and each time this occurs, you can call `OSTimer()`. This results in a system tick period equal to the timer overflow period. You can lengthen this period by using Salvo's timer prescalar. As long as you check often enough not to miss an overflow, you'll have an accurate system timer.

See *How can I avoid re-initializing Salvo's variables when I wake up from sleep on a PIC12C509 PICmicro MCU?*, above, for an example of how to do this.

Context Switching

How do I know when I'm context switching in Salvo?

All Salvo functions implemented as macros (i.e. with an underscore "_" as their third character, e.g. `OS_Yield()`) cause or may cause a context switch. Context switches do not occur anywhere else in Salvo.

Why can't I context switch from something other than the task level?

Because Salvo is designed to run on processors with minimal amounts of RAM memory and no general-purpose stack, it does not presume that a stack is available to store context-switching information. Without it, there's no way to store the return addresses for the function calls nested within the task. If you were to context-

switch from a function nested within a task, upon returning from that function the processor's program counter would be undefined.

Why does Salvo use macros to do context switching?

Context switching in Salvo is an inherently in-line action, and is not generally conducive to the use of functions or subroutines. The context-switching macros use function calls wherever possible to keep code size to a minimum.

Can I context switch in more than one place per task?

There is no limit on how many context switches you write into a given task.

For example, you could add several unconditional context switches (`OS_Yield()`) to the main loop of a low-priority yet long (in terms of lines of code) task. This way, if a higher-priority task needs to run, it will have several opportunities to run for each full path taken through the low-priority task's loop. For example,

```
void TaskLong( void )
{
  for (;;) {
    ...
    /* give other tasks a chance to run.       */
    OS_Yield(TaskLong1);
    ...
    /* let's take a break to let higher-       */
    /*  tasks run.                             */
    OS_Yield(TaskLong2);
    ...
    /* we're about to hog the processor for a  */
    /*  while, so let's yield in case another  */
    /*  more important task is ready to run.   */
    OS_Yield(TaskLong3);
    ....
  }
}
```

When must I use context-switching labels?

Salvo generally requires that you use a unique label for each context switch. The user macro `_OSLabel()` is provided to simplify the declaration of context-switching labels. Some compilers have facilities that make it unnecessary to specify a label as part of a

Salvo context switch. See the documentation specific to your compiler for more details.

If you plan on developing with Salvo across multiple platforms using different compilers, you may find it simplest to use `_OSLabel()` and unique labels for each context switch. The labels will be ignored by those compilers that don't need them.

Tasks & Events

What are taskIDs?

TaskIDs are just integers used to refer to a task. They are numbered from 1 to `OSTASKS`. There's a one-to-one mapping between a task's taskID and the task control block (tcb) assigned to it. You'll rarely use taskIDs when writing your Salvo application. Instead, Salvo uses pointers as handles to tasks. For example, the pointer to the task with taskID 3 is `OSTCBP(3)`.

Does it matter which taskID I assign to a particular task?

No. The only rule to follow is that each task needs its own, unique taskID, and hence its own, unique tcb. A task's priority is independent of its taskID.

Is there an idle task in Salvo?

Salvo has a built-in facility for automatically calling a user-defined function when the system is idling. `OSIdlingHook()` is enabled via the configuration option `OSENABLE_IDLING_HOOK`.

If you prefer, you can create your own idle task with the lowest possible priority (`OSLOWEST_PRIO`). Be sure that no other tasks have this priority. Then, your idle task will run whenever none of the other tasks are eligible.

You can context-switch inside an idle task of your own making, but you cannot context-switch inside the built-in idling hook function. This is an important distinction. Which one you use will depend on what sort of functionality you want to occur when the system is idling. The scheduler must perform a context switch each time the idle task runs. Overall performance is better when using

the idling hook function, since no real context switch is performed when calling `OSIdlingHook()`.

How can I monitor the tasks in my application?

Salvo provides a task monitor function that you can link to your application. The monitor is intended to work with a simple ASCII terminal program. The monitor can display the status of all tasks and events, and can control tasks. See `OSRpt()` for more information.

What exactly happens in the scheduler?

Salvo's scheduler `OSSched()` performs three major functions each time it is called. First, it processes the event queue, if events are in use. This means that for every event that had a waiting task when it was signaled, the scheduler makes that task eligible to run. Next, it processes the delay queue. Any tasks that timed out while being delayed or waiting with a timeout will be made eligible to run. Finally, the scheduler runs the most eligible task. Interrupts are enabled and disabled at various times in the scheduler.

What about reentrant code and Salvo?

An RTOS requires a call…return stack, but Salvo works without a general-purpose stack. Therefore none of its functions are reentrant. In order to avoid problems with reentrancy, 1) do not call a task from anywhere within your program – let the scheduler handle it, and 2) carefully observe the restrictions on calling Salvo services from ISRs. By explicitly controlling interrupts and/or setting certain configuration parameters, you can call certain Salvo services from main-line, task and interrupt levels all in a single application.

What are "implicit" and "explicit" OS task functions?

The explicit OS functions require that you specify a task number as a parameter. A good example is `OSCreateTask()`, which creates and starts a specified task. Explicit OS task function names contain the word "Task". Implicit OS functions like `OS_Delay()` operate only on the current task, i.e. the task that is running. Once a task is running, most or all of the OS functions called are likely to be implicit ones, i.e. they operate on the current task.

How do I setup an infinite loop in a task?

A simple way in C is to use the following syntax:

```
void Task ( void )
{
  /* initialization code. */
  ...

  for (;;) {
    /* body of task. */
    ...
  }
}
```

Note that somewhere in the for loop the task needs to return to the scheduler (e.g. via `OS_Yield()`) to make the highest-priority eligible task run.

Why must tasks use static variables?

Static variables are assigned their own unique address in RAM, and may not be visible to other tasks. By declaring a task's variables as static you are guaranteeing that they will remain unchanged while the task is not running. This is the only way to preserve the variable from one context switch to the next. If the variable were not static (i.e. if it were an auto variable) it's likely that it would be changed by other tasks, functions or ISRs, and unpredictably.

It is safe to use auto variables in tasks[75] as long as the task does not require that the value of the variable be maintained in the task from one context switch to the next. For example, if a simple `for()` loop is used to repeatedly call a function, and then the task context switches, as long as the loop index is initialized each time, it should not pose a problem.

```
int i;

for (;;) {
  for ( i = 0; i < 5 ; i++ ) {
    WriteControlReg(0x55);
    WriteControlReg(0xAA);
  }
  ...
  OS_Yield(here);
}
```

[75] Some implementations (e.g. Salvo on x86-based machines with the Mix Software Power C compiler) do not permit the use of auto variables.

Can tasks share the same priority?

When Salvo is configured to use queues, there's no reason why more than one task cannot share the same priority. Tasks of equal priority will round-robin (execute one after the another in a circular queue) whenever they are the highest-priority eligible tasks. However, in many applications it is more efficient to give each task a unique priority.

When Salvo is configured to use arrays, each task must have a unique priority.

If an idle task is used in your Salvo application, it should be the only task with the lowest priority (`OSLOWEST_PRIO`). Other tasks should use priorities between `OSHIGHEST_PRIO` and `OSLOWEST_PRIO-1`.

Can I have multiple instances of the same task?

Yes. A Salvo task is essentially an address in your program at which your application will resume execution when the scheduler sends it there. You can configure two or more Salvo tasks to point to the same place in your program. For example,

```
void TaskDelayFiveTicks( void )
{
  for (;;)
    OS_Delay(5, here);
}

...
OSCreateTask(TaskDelayFiveTicks, OSTCBP(5), 8);
OSCreateTask(TaskDelayFiveTicks, OSTCBP(6), 9);
...
for (;;)
  OSSched();
```

will create two Salvo tasks with different priorities, each of which delays itself for 5 system ticks over and over. Note that without reentrancy, the utility of multiple instances of the same task is limited. Note also that all static variables in the task function will be "shared" by each instance of Salvo task.

Does the order in which I start tasks matter?

No. To start a task, it must have been created first. Creating a task initializes the fields in its task control block, but leaves it ineligible

to run. Starting a task makes it eligible and places it in the eligible queue. Tasks are positioned within the eligible queue based on their priority. A task will first execute based on its priority, not on when it was started.

If you start several tasks of equal priority together, they will begin executing in the order they were started. If they remain at these same priorities, they will continue to round robin.

By using `OSSetPrio()` or `OS_Prio()` to change the current task's priority you can control the order in which tasks execute.

How can I reduce code size when starting tasks?

You may face this question of you are explicitly starting tasks separately from when they are created (by using `OSDONT_START_TASK` with `OSCreateTask()`). Each task is referred to by its tcb pointer, which is specified in the call to `OSCreateTask()`. You can reduce the number of calls to `OSStartTask()` by placing it in a loop in order to start multiple tasks at once, e.g.

```
char i;
...
for ( i = 1 ; i <= OSTASKS ; i++ )
  OSStartTask(OSTCBP(i));
```

will start all of your tasks with just a single call to `OSStartTask()`, thereby reducing the size of your application.

What is the difference between a delayed task and a waiting task?

A task that is delayed is simply inactive for a specified number of system ticks. It will then rejoin the eligible tasks when the delay timer has expired. A task that is waiting will wait until an event occurs. If the event never occurs, then the task is never made eligible again, unless a timeout was specified when the task was made to wait. If the timeout timer expires before the event occurs, the task is made eligible and carries with it a flag that indicates that a timeout occurred. Your application program can handle this flag at the task level.

In order to delay tasks, `OSTimer()` must be called at the system tick rate from your application. This run-time overhead is independent of the number of tasks still delayed. Waiting tasks, on the

other hand, do not require the existence of `OSTimer()`,[76] and require no processing power whatsoever while they are waiting.

Can I create a task to immediately wait an event?

Not with a single service call. A task can only wait an event by calling `OS_WaitXyz()` while running. One way to start your application with a bunch of tasks waiting for event(s) is to create them with the highest priority (guaranteeing that they will run before all others) and create the events with initial values of 0. When each task runs, have it change its priority to the desired run-time priority with `OSSetPrio()` (not `OS_Prio()`!), and have it wait the event. When the events are signaled, the waiting tasks will run.

I started a task but it never ran. Why?

You may have incorrectly specified one or more parameters when calling the relevant Salvo services – check the function return codes to see if any errors were reported. A common error when using the freeware libraries is to create a task with a tcb pointer that exceeds `OSTCBP(OSTASKS)`.

If the OS was initialized via `OSInit()`, the task was successfully created and started via `OSCreateTask()`, the scheduler `OSSched()` is active, and no other task has destroyed or stopped the task in question, then it probably had a lower priority than the other tasks running, and hence never ran. Try elevating the task's priority. Use the Salvo monitor `OSRpt()` to view the current status of all the tasks.

What happens if I forget to loop in my task?

You'll get some rather odd results. If your application doesn't crash immediately, the original task may leave its own function and continue through your code until it reaches a context switch, and will thereafter resume execution after that context switch, which will be part of another task! So you may have inadvertently created a second instance of another task by failing to keep execution within the intended task.

---

76 Unless they were made to wait with a timeout.

Why did my low-priority run-time tasks start running before my high-priority startup task completed?

It's common to use delays in a startup task (responsible for configuring peripherals like LCDs, for instance). The other tasks ran because the high-priority startup task was delayed. Regardless of its priority, whenever a task is delayed or waiting for an event, other lower-priority tasks are free to run.

If your application needs a startup task that uses delays, and if it's imperative that no other tasks run before the startup task is complete, then one elegant method is to initially create all the tasks but only start the startup task, and then start the other tasks at the end of the startup task. You can even "reuse" the startup task's tcb by destroying the startup task and creating a new task with the same tcb.

When I signaled a waiting task, it took much longer than the context switching time to run. Why?

A task that is made eligible will only run when it becomes the highest-priority eligible task. Other eligible tasks with higher priorities will run first, and will continue to run if they remain eligible. Also, interrupt service routines (ISRs) have the highest priorities of all.

Can I destroy a task and (re-) create a new one in its place?

Yes. As long as a task is destroyed, a new one can be created in its place. A Salvo task is really just a means of executing a function in ROM. Creating and starting a task allows that function to execute along with the other tasks in a priority-based scheme.

Before destroying any task you must ensure that:

- it is not waiting for any event,
- is it in the delayed queue and
- has not acquired any resources that other tasks might need.

It is up to you to ensure that the above conditions are met. If you are to use `OSDestroy()` in a particular task that accesses resources, you must release all resources before destroying the task. Failing to do so would block any other tasks waiting for the resource previ-

ously owned by the now-destroyed task. Only if those tasks were waiting with a timeout would they ever run again.

Can more than one task wait on an event?

Yes. Up to all of the defined tasks can wait on a single event simultaneously.

Does Salvo preserve the order in which events occur?

Yes.

Can a task wait on more than one event at a time?

Yes, but not simultaneously. At any time a task can only be waiting on a single event. It can wait on more than one event sequentially (e.g. first on one, then on the other), but not simultaneously.

In this example, a task first waits for an error message (a string), then waits for a resource (an LCD display) to become available. Once it receives the error message and obtains exclusive access to the display, it writes the message to the display, waits one second, releases the display for others to use, and then returns to waiting for another message.

```
void TaskShowErrMsg( void )
{
  static OStypeMsgP msgP, msgP2;

  for (;;) {
    OS_WaitMsg(MSG_ERROR_STRING_P, &msgP,
      OSNO_TIMEOUT, label);
    OS_WaitMsg(MSG_LCD_DISPLAY_P, &msgP2,
      OSNO_TIMEOUT, label2);
    DispStringOnLCD((char *) msgP);
    OS_Delay(ONE_SECOND, label3);
    OSSignalMsg(MSG_LCD_DISPLAY_P, (OStypeMsgP)
1);
  }
}
```

By first acquiring the display resource and later releasing it,[77] the user is guaranteed to see the error message for at least one second. The error message will remain on the LCD display until this or another task obtains the LCD display resource via `OS_Wait-`

[77] In this example, `MSG_LCD_DISPLAY` is being used as a binary semaphore.

`Msg(MSG_LCD_DISPLAY, ...)` and writes a new string to it via `DispStringOnLCD()`.

How can I implement event flags?

Event flags are used to synchronize tasks to the occurrence of multiple events. Two types of synchronization are possible – *conjunctive synchronization*, where the task can only proceed once all of the events it's waiting on have occurred (i.e. logical AND), and *disjunctive synchronization*, where the task can proceed as soon as any of the events it's waiting on has occurred (i.e. logical OR).

You can use Salvo's built-in event flag (eFlag) services(this is the preferred method), or you can implement simple flags using binary semaphores. See the Reference chapter in the *Salvo User Manual* for more info on Salvo's event flag services.

To implement conjunctive synchronization (i.e. the logical AND of multiple events) using binary semaphores, the task must wait on multiple events in sequential order. In the example below, the task waits for the occurrence of all three events (signified by binary semaphores) before proceeding.

```
…
OS_WaitBinSem(BINSEM1_P, OSNO_TIMEOUT,
  WaitForSync1);
OS_WaitBinSem(BINSEM2_P, OSNO_TIMEOUT,
  WaitForSync2);
OS_WaitBinSem(BINSEM3_P, OSNO_TIMEOUT,
  WaitForSync3);
…
```

The order in which the events occur (i.e. when each event is signaled) is unimportant. As long as the task is the highest-priority task waiting on each event, once all of the events have been signaled the task will proceed.

To implement disjunctive synchronization (i.e. the logical OR of multiple events) using binary semaphores, the task must wait on a single event that can be signaled from multiple locations in your application.

```
…
OS_WaitBinSem(BINSEM4_P, OSNO_TIMEOUT,
  WaitForSync4);
…
```

In this case the task can proceed as soon as any part of your application has signaled the event. Subsequent event signaling will not affect the task's execution until the next time it waits on the event.

What happens when a task times out waiting for an event?

If the task does not acquire the resource within the timeout period, it will be removed from the event queue (and the waiting queue) and made eligible to run again. When it runs, a timeout flag will be available at the task level to indicate that a timeout occurred. The Salvo user service `OSTimedOut()` returns `TRUE` when this flag is set, `FALSE` otherwise. The timeout flag is cleared when the task returns to the scheduler.

If a task times out waiting for an event, even if the event subsequently occurs before the task runs again, the timeout flag will remain until the task runs and returns to the scheduler. The event will also remain until a task waits on it.

Why is my high-priority task stuck waiting, while other low-priority tasks are running?

The unavailability of an event always takes precedence over a task's priority. Therefore, regardless of its priority, a task that waits on an event that is not available will become a waiting task, and it will remain a waiting task until either a) the event happens and the task is the highest-priority task waiting for the event, or b) a timeout (if specified) occurs.

This situation may simply be due to the fact that the event never occurred, or it may be due to priority inversion.

When an event occurs and there are tasks waiting for it, which task(s) become eligible?

The highest-priority waiting task becomes eligible. Only a single task will become eligible, regardless of how many tasks of equal priority are waiting for the event. All of Salvo's queues are priority queues. Additionally, tasks of equal priorities are inserted into the priority queues (i.e. they are enqueued) on a FIFO basis. For example, if a task of the highest priority is enqueued into a priority queue that already contains a task of highest priority, the task being enqueued will be enqueued after the existing task. In other words,

the first task to be enqueued with a particular priority will be the first task to be dequeued when tasks of that particular priority reach the head of the queue.

How can I tell if a task timed out waiting for an event?

The macro `OSTimedOut()` is provided to detect timeouts. It returns `TRUE` if the current task has timed out waiting for an event, and `FALSE` otherwise. `OSTimedOut()` is only valid while the current task is running.

Can I create an event from inside a task?

Yes. You can create an event or a task anywhere in your code, as long as you have previously allocated the required memory at compile time. Keep in mind that operating on an event that is not yet defined can cause unpredictable behavior. For example, suppose you have two tasks, one to create and signal a resource, and one that waits for it:

```
void Task1( void )
{
  OSCreateSem(SEM1_P, 0);  /* init to 0 */

  for (;;) {
    ...
    OSSignalSem(SEM1_P);
    ...
  }
}

void Task2( void )
{
  for (;;) {
    ...
    OS_WaitSem(SEM1_P, OSNO_TIMEOUT, label);
    ...
  }
}
```

If your `main()` looks like this:

```
int main( void )
{
  OSInit();
  OSCreateTask(Task1, TASK1_P, 3);
  OSCreateTask(Task2, TASK2_P, 1);
  for (;;)
    OSSched();
}
```

you will have unpredictable results because `Task2()` will attempt to wait the semaphore `SEM1` before `Task1()` can create it. That's because `Task2()` has a higher priority than `Task1()`, and will therefore run first when the `OSSched()` starts dispatching tasks.

To avoid this, you can either ensure that the task that creates the resource has a higher priority than any task that uses it, or you can create the resource before beginning multitasking via `OSSched()`.

If you plan on creating events or tasks from within an ISR, you must configure `salvocfg.h` appropriately to avoid interrupt-related issues.

What kind of information can I pass to a task via a message?

Messages are application-specific – that is, a message contains whatever you want it to contain. Examples include characters, numbers, strings, structures and pointers. Messages are passed via pointer, and the default type for a Salvo message pointer is `OStypeMsgP`, which is usually a void pointer. Since a void pointer can point to anything, in order to obtain the information in the message, you'll need to typecast the pointer's contents to the message's inherent type.

The only restriction on Salvo messages is that all the messages in a particular message queue should point to the same type of information.

My application uses messages and binary semaphores. Is there any way to make the Salvo code smaller?

Yes, use messages with values of `(OStypeMsgP) 0` and `(OStypeMsgP) 1` instead of binary semaphores with values of `0` and `1`, respectively. This way you can use `OSCreateMsg()`, `OSSignalMsg()` and `OS_WaitMsg()` exclusively.

Why did RAM requirements increase substantially when I enabled message queues?

Each message queue requires both an ecb and a message queue control block (mqcb) of fixed size. The number of ecbs and mqcbs

are determined by OSEVENTS and OSMESSAGE_QUEUES, respectively. Additionally, each message queue also requires RAM for the actual queue. Message queues are the only events that require this extra memory.

Can I signal an event from outside a task?

Yes. Events can be signaled and created from mainline code (e.g. from within tasks, functions or inside main()), and from within interrupts. The default Salvo configuration expects events to be created and signaled from mainline code. In order to create or signal tasks from interrupts and/or interrupts and mainline code, the configuration parameters appropriate to the event's user service (e.g. OSSignalMsg()) must be defined.

When I signal a message that has more than one task waiting for it, why does only one task become eligible?

A task waits for a message when the corresponding mailbox is empty. Signaling a message will fill the mailbox. The mailbox remains full (i.e. contains a single message) until the task that was waiting on the message runs, i.e. until the task becomes the highest-priority task and is dispatched by the scheduler. Put another way, signaling a message fills the mailbox, and running the task that's waiting on the message empties it. If the task never becomes eligible to run, the mailbox will remain full, and signaling it with a message will result in an error.

I'm using a message event to pass a character variable to a waiting task, but I don't get the right data when I dereference the pointer. What's going on?

Let's say you're trying to pass a character to a task via a message. To send the message you might write:

```
char tempVar;
...
tempVar = '!';
OSSignalMsg(MSG_CHAR_TO_TASK_P,
  (OStypeMsgP) &tempVar);
...
```

to send a '!' to the task that's waiting for the message MSG_CHAR_TO_TASK, which might look like this:

```
static OStypeMsgP msgP;
static char msgReceived;

for (;;) {
  OS_WaitMsg(&msgP, MSG_CHAR_TO_TASK_P,
    OSNO_TIMEOUT, label);
  msgReceived = *(char *) msgP;
  switch ( msgReceived ) {
    case '!':
      printf("Received '!'\n");
      break;

    default:
      printf("Received anything but '!'\n");
  }
}
```

Because tasks obtain messages via pointers, the element referenced by the message pointer must remain unchanged until `OS_WaitMsg()` succeeds. In the example above, if the global or auto variable `tempVar` is assigned another value before the waiting task has a chance to obtain the message, the waiting task will receive a message quite different from what was intended. A safer solution would be to signal the message with a pointer to a character constant:

```
const char BANG = '!';
...
OSSignalMsg(MSG_CHAR_TO_TASK_P,
  (OStypeMsgP) &BANG);
...
```

This way, no matter how long it takes for the receiving task to run and obtain the message, it is guaranteed to be the '!' character.

What happens when there are no tasks in the eligible queue?

The scheduler loops in a very tight loop, with interrupts enabled, when there are no tasks eligible to run. As soon as a task is made eligible, either through the actions of `OSTimer()` or an interrupt signaling an event, the scheduler will cause it to run.

In what order do messages leave a message queue?

Each message queue operates on a FIFO (first-in, first-out) basis.

What happens if an event is signaled before any task starts to wait it? Will the event get lost or it will be processed after task starts to wait it?

The event will not be lost, and the highest-priority task to wait the event will get it, i.e. will remain eligible after `OS_WaitXyz()` instead of going to the waiting state.

What happens if an event is signaled several times before waiting task gets a chance to run and process that event? Will the last one signal be processed and previous lost? Or the first will be processed and the following signals lost?

That depends on the event – if it's a binary semaphore or a message, all further signaling results in `OSSignalXyz()` returning an error code, because the event is "full". The first event to be signaled will be processed, and subsequent ones will be lost. In the case of a counting semaphore, the value is simply incremented. In the case of a message queue, additional messages are enqueued until the queue is full. With these events, once the event is "full", subsequent signals will be lost.

What is more important to create first, an event or the task that waits it? Does the order of creation matter?

The order of creation doesn't matter. But when a task waits an event, the event must exist before the task runs.

What if I don't need one event anymore and want to use its slot for another event? Can I destroy event?

Absolutely! For example, you can destroy a binary semaphore and create a counting semaphore in its place by calling `OSCreateSem()` with the ecb you previously used for the binary semaphore. You should only do this if you know that there aren't any tasks waiting the binary semaphore.

Can I use messages or message queues to pass raw data between tasks?

Yes, with some restrictions. With messages, a null message pointer is treated as an empty message, and a task will wait an empty message forever. Therefore only non-zero raw data can be passed via messages. Message queues are different in that a task will wait a message queue indefinitely if there are no messages in it. Therefore null message pointers are allowed in message queues, and raw data of any value can be passed from one task to another using a message queue. In this case, the message queue acts like a FIFO buffer.

If you want to pass null-pointer messages to a task, use a message queue of size 1.

How can I test if there's room for additional messages in a message queue without signaling the message queue?

Use `OSMsgQEmpty()`. If the message queue is full – i.e. there is no room for an additional message in the message queue – `OSMsgQEmpty()` returns 0 (FALSE). If there is room, `OSMsgQEmpty()` returns the number of available slots in the message queue.

Interrupts

Why does Salvo disable all interrupts during a critical section of code?

It is common practice in an RTOS to disable interrupts during a critical section of code. To maintain system performance, interrupts should be disabled for the shortest times possible. However, it's imperative that while an RTOS performs certain critical functions, it must not be interrupted for fear of certain things in the RTOS being corrupted.

The major sources of corruption due to interference from an interrupt are access to a shared resource, and the operation of non-reentrant functions. Salvo must guarantee that while performing certain operations on its data structures (e.g. changing an event control block), no access (read or write) from any other part of the application is allowed. Salvo functions that access the data struc-

tures include `OSTimer()`, which is normally called from within a periodic interrupt, and `OSSignalMsg()`, which might be called from an entirely different interrupt.

Since Salvo services work without a general-purpose stack, certain steps must be taken to prevent data corruption from interrupts. Use the `OSCALL_Xyz()` configuration parameters if you want to be able to call a particular Salvo service (e.g. `OSSignalSem()`) from both main-line code and an ISR.

Can I modify Salvo to disable only certain interrupts during critical sections of code?

Loosely speaking, if the only Salvo function called by an interrupt is `OSTimer()`, and in no other place in your application do you call any function (internal to Salvo, compiler library or otherwise) used by Salvo, then you may get by modifying the `OSDi()` and `OSEi()` macros to disable only the source of interrupts that results in a call to `OSTimer()`, leaving all other interrupts enabled during critical sections of code. For example, on the Microchip PIC17-series PICmicro devices you may be able to disable peripheral interrupts while leaving global interrupts enabled. However, this is not recommended, as it's very difficult to ensure that the second condition above is met, and it's also possible to forget these restrictions over time and violate them at a later date when your application is modified. As a result, this technique is not recommended.

How big are the Salvo functions I might call from within an interrupt?

`OSTimer()` and `OSSignalXyz()` are the Salvo services you might call from an interrupt. They are all quite small and fast, and have no nested subroutines. While it varies among different target processors, these services will in many cases be faster than the actual interrupt save and restore.

Why did my interrupt service routine grow and become slower when I added a call to OSTimer()?

Some compilers assume the worst case with regard to register saves and restores when an external function is called from within an interrupt routine. As a result, the compiler may add a large amount of code to save and restore registers or temporary registers

to preserve the program's context during an interrupt. Since it's always a good idea to have as fast an interrupt routine as possible, one solution is to include the necessary Salvo files[78] in your interrupt routine's source code instead of linking to the `OSTimer()` and related services as external functions (e.g. through the Salvo library). By including those Salvo files which completely define the necessary call chains for `OSTimer()` your compiler can "see" exactly which registers and temporary registers must be saved, instead of assuming the worst case and saving all of them.

Another option is to in-line OSTimer(). For more information, see the `OSUSE_INLINE_OSTIMER` configuration option.

My application can't afford the overhead of signaling from an ISR. How can I get around this problem?

Ideally you should signal from an ISR if the event that causes the signaling is an interrupt. If this is not possible, in your ISR you can set a simple flag (i.e. a bit) in a global variable, and then test-and-clear it[79] in your main loop. If the flag is set, you then call the appropriate signaling service prior to calling `OSSched()`, like this:

```
for (;;) {
  di();
  localFlag = flag;
  flag = 0;
  ei();
  if ( localFlag ) OSSignalBinSem(binSemP);
  OSSched();
}
```

This disadvantage of this approach is that it does not preserve the order in which events occur, whereas signaling from an ISR will preserve that order. This may affect the behavior of complex systems.

78 `timer.c`.

79 Interrupts should be disabled while you test and clear the flag.

Building Projects

What warning level should I use when building Salvo projects?

Use the compiler's default warning level. More pedantic warning levels may generate warnings that in some cases cannot be avoided, and thus cause unnecessary confusion.

What optimization level should I use when building Salvo projects?

Use the maximum optimization unless suggested otherwise.

Miscellaneous

Can Salvo run on a 12-bit PICmicro with only a 2-level call…return stack?

Yes. Certain compilers (e.g. HI-TECH PICC) circumvent this limitation by converting all function calls into long jumps through table lookup. Therefore function calls require some additional overhead and ROM, but call graphs of arbitrary depth are possible.

Chapter 7 • Reference

User Services

This section describes the Salvo user services that you will use to build your multitasking application. Each user service description includes information on:

- the service *type* (function or macro),
- the service *prototype* (for a function) or *declaration* (for a macro),
- where the service is *callable from* (the foreground, the background or within a task),
- which Salvo C source or include files *contain* the source code for the service,
- which configuration options (if any) *enable* the service,
- which configuration options (if any) *affect* the service (i.e. alter its execution speed or code size),
- a *description* of what the service does,
- the *parameter(s)* (if any) expected by the service call,
- the service's *return* value(s) (if any),
- the service's *stack usage* (if any), in terms of levels of call…return stack used,[80]
- *notes* particular to the service,
- *related* services and
- an *example* using the service.

Salvo functions comprise the majority of the user services you will call from C in your application. Salvo user services that do not result in a context switch are implemented as functions and are prefixed by "`OS`".

Salvo uses macros wherever a context-switch is implicit in the action being performed (e.g. delaying for a number of ticks, via `OS_Delay()`). All of Salvo's services that result in a context-switch are implemented via macros and are prefixed by "`OS_`".

---

[80] For call…return stack depth calculations, `OSUSE_INSELIG_MACRO` is assumed to be the default value, `TRUE`. If `FALSE`, those services that cause a task to be placed in the eligible, delay and/or event queue(s) will consume an additional call…return stack level. Stack usage does not take into account any library functions invoked by the compiler.

It is important not to confuse a Salvo macro with its underlying function. For instance, the `OS_Delay()` macro will cause the current task to delay for the specified number of system ticks. On the other hand, using the `OSDelay()` function directly will have unpredictable results, and your application may crash as a result. These underlying functions are intended for use only within a Salvo macro, and are therefore not documented in this section. For the curious, they can be viewed in the Salvo source code.

When compiling and linking Salvo into your application, the size and speed of many user services is dependent on the chosen configuration. By referring to the detailed descriptions of each user service below and inspecting the output of your compiler, you may be able to correlate changes in the size (in instructions) and/or speed (in cycles) of the Salvo services in your application against changes you've made to your compile-time configuration. Remember that each time you change the configuration options, you must recompile all of Salvo before linking it into your application.

Note The *foreground* is the interrupt level of your application. The *background* is the non-interrupt level, and includes `main()`, Salvo tasks and all other functions not called via interrupts.

Macro Declarations

The macro declarations for Salvo's user functions are listed below.

```
OS_Delay     ( OStypeDelay  delay,
               label );

OS_DelayTS   ( OStypeDelay  delay,
               label );

OS_Destroy   ( label );

OS_Prio      ( OStypePrio   prio,
               label );

OS_Replace   ( OStypeTFP    tFP,
               OStypePrio   prio );

OS_Stop      ( label );

OS_WaitBinSem( OStypeEcbP   ecbP,
               OStypeDelay  timeout,
               label );

OS_WaitEFlag ( OStypeEcbP   ecbP,
               OStypeEFlag  mask,
               OStypeOption options,
```

```
                OStypeDelay  timeout,
                label );

OS_WaitMsg    ( OStypeEcbP   ecbP,
                OStypeMsgPP  msgPP,
                OStypeDelay  timeout,
                label );

OS_WaitMsgQ   ( OStypeEcbP   ecbP,
                OStypeMsgPP  msgPP,
                OStypeDelay  timeout,
                label );

OS_WaitSem    ( OStypeEcbP   ecbP,
                OStypeDelay  timeout,
                label );

OS_Yield      ( label );

OSTimedOut    ( );

OSVersion     ( );
```

Note Salvo context-switching services are implemented as macros and do not have return.

Function Prototypes

The function prototypes for Salvo's user functions are listed below.

Note Some services (e.g. `OSCreateXyz()` and `OSSignalXyz()`) can be either a macro that invokes a function, or a standalone function, depending on `OSCOMBINE_EVENT_SERVICES`. In all cases the argument list and return value and type are identical.

```
OStypeErr     OSClrEFlag     ( OStypeEcbP      ecbP,
                               OStypeEFlag     mask );

OStypeErr     OSCreateBinSem ( OStypeEcbP      ecbP,
                               OStypeBinSem    binSem );

OStypeErr     OSCreateEFlag  ( OStypeEcbP      ecbP,
                               OStypeEfcbP     efcbP,
                               OStypeEFlag     eFlag );

OStypeErr     OSCreateMsg    ( OStypeEcbP      ecbP,
                               OStypeMsgP      msgP );

OStypeErr     OSCreateMsgQ   ( OStypeEcbP      ecbP,
                               OStypeMqcbP     mqcbP,
                               OStypeMsgQPP    msgPP,
                               OStypeMsgQSize  size );

OStypeErr     OSCreateSem    ( OStypeEcbP      ecbP,
                               OStypeSem       sem );
```

```
OStypeErr    OSCreateTask   ( OStypeTFP       tFP,
                              OStypeTcbP      tcbP,
                              OStypePrio      prio );

OStypeErr    OSDestroyTask  ( OStypeTcbP      tcbP );

OStypePrio   OSGetPrio      ( void );

OStypePrio   OSGetPrioTask  ( OStypeTcbP      tcbP );

OStypeState  OSGetState     ( void );

OStypeState  OSGetStateTask ( OStypeTcbP      tcbP );

void         OSIdlingHook   ( void );

void         OSInit         ( void );

OStypeMsgQSize  OSMsgQCount ( OStypeEcbP      ecbP );

OStypeMsgQSize  OSMsgQEmpty ( OStypeEcbP      ecbP );

OStypeTick   OSGetTicks     ( void );

OStypeBinSem OSReadBinSem   ( OStypeEcbP      ecbP );

OStypeEFlag  OSReadEFlag    ( OStypeEFlag     ecbP );

OStypeMsgP   OSReadMsg      ( OStypeEcbP      ecbP );

OStypeMsgP   OSReadMsgQ     ( OStypeEcbP      ecbP );

OStypeSem    OSReadSem      ( OStypeEcbP      ecbP );

void         OSRpt          ( void );

void         OSSched        ( void );

OStypeErr    OSSetEFlag     ( OStypeEcbP      ecbP,
                              OStypeEFlag     mask );

void         OSSetPrio      ( OStypePrio      prio );

OStypeErr    OSSetPrioTask  ( OStypeTcbP      tcbP,
                              OStypePrio      prio );

void         OSSetTicks     ( OStypeTick      tick );

OStypeErr    OSSignalBinSem ( OStypeEcbP      ecbP );

OStypeErr    OSSignalMsg    ( OStypeEcbP      ecbP,
                              OStypeMsgP      msgP );

OStypeErr    OSSignalMsgQ   ( OStypeEcbP      ecbP,
                              OStypeMsgP      msgP );

OStypeErr    OSSignalSem    ( OStypeEcbP      ecbP );

OStypeErr    OSStartTask    ( OStypeTcbP      tcbP );

OStypeErr    OSStopTask     ( OStypeTcbP      tcbP );
```

```
void         OSTimer        ( void );
OStypeBinSem OSTryBinSem    ( OStypeEcbP      ecbP );
OStypeMsgP   OSTryMsg       ( OStypeEcbP      ecbP );
OStypeMsgP   OSTryMsgQ      ( OStypeEcbP      ecbP );
OStypeSem    OSTrySem       ( OStypeEcbP      ecbP );
```

OS_Delay(): Delay the Current Task and Context-switch

Type:	Macro (invokes `OSDelay()`)
Declaration:	`OS_Delay ( OStypeDelay delay, label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	`OSBYTES_OF_DELAY`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSLOGGING`
Description:	Delay the current task by the amount specified. Return to scheduler.
Parameters:	`delay`: an integer (>=0) specifying the desired delay in system ticks. `label`: a unique label.
Returns:	–
Stack Usage:	2

Notes

A delay of 0 will stop the current task. A non-zero delay will delay[81] the current task by the number of ticks specified relative to the current value of the system timer.

Do not call `OS_Delay()` from within an ISR!

In order to use delays, Salvo's timer must be installed.

Long delays can be accomplished in a variety of ways – See "Timer and Timing" in *Chapter 6 • Frequently Asked Questions (FAQ)*.

In the example below (system tick rate = 40Hz, t = 25ms, Hitachi 44780 LCD controller), `OS_Delay()` is used to delay the LCD task `TaskDisp()` during startup while the LCD is being configured. By using `OS_Delay()` instead of an in-line delay, the other tasks may run while `TaskDisp()` is delayed and the LCD is initialized.

See Also

`OS_DelayTS()`, `OS_Stop()`, `OSTimer()`

[81] When delaying a task repetitively, remember that there is an additional, unpredictable delay between when the task's delay expires and when it actually runs. This may happen if there are other, higher-priority tasks eligible to run when the delayed task's delay expires. This can affect a task's "loop delay."

Example

```
#define LCD_CMD_REG    0    /* for commands    */
#define LCD_DATA_REG   1    /* for data        */
#define LCD_CMD_CLS    0x01 /* clear display   */
#define LCD_CMD_MODE   0x06 /* auto-inc address*/
#define LCD_CMD_ON_OFF 0x0C /* on, no cursor,  */
                            /*  no blink       */
#define LCD_CMD_FN_SET   0x3F
#define LCD_BITMASK_RS   0x01 /* reg select    */
#define LCD_BITMASK_RW   0x02 /* read/-write   */
#define LCD_BITMASK_E    0x04 /* E (strobe)    */

void TaskDisp( void )
{
  static OStypeMsgP msgP;

  /* initialize the LCD Display                */
  char i;         /* doesn't need to be static */

  TRISD = 0x00; /* all LCD ports are outputs   */
  TRISE = 0x00; /*  "                          */
  PORTE = 0x00; /* RS=0, -WRITE, E=0           */

  /* we want to talk to the command register,  */
  /*  and we'll wait 50ms to ensure it's       */
  /*  listening.                               */
  LCDSelReg(LCD_CMD_REG);
  OS_Delay(2, TaskDisp1);

  /* Hitachi recommends 4 consecutive writes   */
  /*  to this register ...                     */
  for ( i = 4 ; i-- ; )
    LCDWrData(LCD_CMD_FN_SET);

  /* configure LCD the "standard" way.         */
  LCDWrData(LCD_CMD_ON_OFF);
  LCDWrData(LCD_CMD_MODE);
  LCDWrData(LCD_CMD_CLS);

  /* wait another 50ms.                        */
  OS_Delay(2, TaskDisp2);

  /* now we're done initializing LCD display.  */
  …

  for (;;) {
    OS_WaitMsg(MSG_UPDATE_DISP_P, &msgP,
      OSNO_TIMEOUT, TaskDisp3);
    …
  }
}
```

OS_DelayTS(): Delay the Current Task Relative to its Timestamp and Context-switch

Type:	Macro (invokes `OSDelay()`)
Declaration:	`OS_DelayTS ( OStypeDelay delay, label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	`OSBYTES_OF_DELAY`, `OSBYTES_OF_TICKS`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSLOGGING`
Description:	Delay the current task by the amount specified, relative to the task's timestamp. Return to scheduler.
Parameters:	`delay`: an integer (>=0) specifying the desired delay in system ticks. `label`: a unique label.
Returns:	–
Stack Usage:	2

Notes

A delay of 0 will stop the current task. A non-zero delay will delay the current task by the number of ticks specified relative to the task's timestamp. The timestamp is automatically recorded by `OSInit()` and whenever a task's delay times out. In order to use delays with timestamps, Salvo's timer must be installed and the counting of system ticks must be enabled via `OSBYTES_OF_TICKS`.

If more than `delay` and less than 2 x `delay` system ticks occur between the task's delay expiring and the task running,[82] the task will attempt to resynchronize itself for the *following* delay period. The behavior for more than 2 x `delay` ticks is undefined.[83]

Do not call `OS_Delay()` from within an ISR!

In the example below, `TaskA()` will always run every fourth system tick because it is synchronized to the system timer. As long as the delay between the task's delay expiring and the task actually running[84] never exceeds 2 `delay` periods, the task will always run at t_0 + (number of iterations x `delay`).

---

82 I.e. the task is " very late".

83 In this situation you may need to chose a longer system tick period.

84 This might happen if, for instance, `TaskA()`'s priority is low, and there are other tasks eligible to run.

See Also

OS_Delay(), OSGetTS(), OSSetTS(), OS_Stop(), OSSyncTS(), OSTimer()

Example

```
void TaskA( void )
{
  while ( TRUE ) {
    OS_DelayTS(4, TaskAlabel);
    …
  }
}

main()
{
  …

  OSInit();

  OSCreateTask(TaskA, OSTCBP(1), 4);
  …

  ei(); /* enable interrupts */

  while ( TRUE ) {
    OSSched();
  }
}
```

OS_Destroy(): Destroy the Current Task and Context-switch

Type:	Macro (invokes `OSDestroy()`)
Declaration:	`OS_Destroy ( label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Destroy the current task. Return to scheduler.
Parameters:	`label`: a unique label.
Returns:	–
Stack Usage:	1

Notes

Once a task is destroyed, it cannot be restarted. However, a new task can be created in its place by using the same tcb.

Do not call `OS_Destroy()` from within an ISR!

In the example below, `TaskStartup()` creates and starts most of the other tasks in the application. `TaskDisp()` (see example for `OS_Delay()`) will run immediately after `TaskStartup()` begins its two-second delay. When the delay expires, `TaskStartup()` will resume, creating and starting `TaskMsg()`, `TaskRdKey()`, `TaskStatus()`, `TaskTx()` and `TaskRx()`. However, none of these tasks will run until `TaskStartup()` destroys itself and returns to the scheduler. Once `TaskRx()` runs it will create `TaskRcvRsp()` in place of `TaskStatus()`, thereby reusing the tcb for another task. `TaskStartup()` is not structured as an infinite loop – rather, it's simply a one-time sequence of events, which ends when `TaskStartup()` destroys itself and returns to the scheduler.

See Also

`OSCreateTask()`, `OSStop()`

Example

```
void TaskStartup( void )
{
  /* create all the tasks we need early on.    */
  /*  Some of these tasks create other tasks   */
  /*  and resources! Start them up, too.       */

  /* TaskDisp() handles display updates. It    */
  /*  also creates MSG_DISP & SEM_UPDATE_DISP. */
  OSCreateTask(TaskDisp, TASK_DISP_P,
    TASK_DISP_PRIO);

  /* Leave startup screen showing for 2s.      */
  OS_Delay(TWO_SEC, TaskStartup1);

  /* TaskMsg() flashes messages. It also       */
  /*  creates MSG_FLASH_STRING.                */
  OSCreateTask(TaskMsg, TASK_MSG_P,
TASK_MSG_PRIO);

  /* TaskRdKey() reads the keypad. It also     */
  /*  creates MSG_KEY_PRESSED and creates and  */
  /*  starts TaskRcvKeys().                    */
  OSCreateTask(TaskRdKey, TASK_RD_KEY_P,
    TASK_RD_KEY_PRIO);

  /* TaskStatus() monitors the PSR on Driver.  */
  /*  It also creates MSG_WAKE_STATUS and      */
  /*  MSG_LONG_OP_DONE.                        */
  OSCreateTask(TaskStatus, TASK_STATUS_P,
    TASK_STATUS_PRIO);

  /* TaskTx() send cmds out to the Driver. It  */
  /*  also creates MSG_WAKE_TX, MSG_RSP_RCVD   */
  /*  and MSG_TX_BUFF_EMPTY.                   */
  OSCreateTask(TaskTx, TASK_TX_P, TASK_TX_PRIO);

  /* TaskRx() receives responses back from the */
  /*  Driver. It also creates SEM_RX_RBUFF and */
  /*  creates and starts TaskRcvRsp().         */
  OSCreateTask(TaskRx, TASK_RX_P, TASK_RX_PRIO);

  /* we're finished starting up, so kill this  */
  /*  task permanently. TaskRcvKeys() will     */
  /*  "take over" its tcb – see                */
  /*  TaskRdKeys().                            */
  OS_Destroy(TaskStartup2);
}
```

OS_Replace(): Replace the Current Task and Context-switch

Type:	Macro (invokes `OSCreateTask()`)
Declaration:	`OS_Replace (tFP, prio);`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	–
Description:	Replace the current task with the one specified. Return to scheduler.
Parameters:	`tFP`: a pointer to the task's start address. This is also the task's function prototype name. `prio`: the desired priority for the task. If OR'd with `OSDONT_START_TASK`, the task will not be started.
Returns:	–
Stack Usage:	3

Notes

The task that replaces the current task will use the same tcb. Once a task is replaced, it can be restarted only with a call to `OSCreateTask()`.

Do not call `OS_Replace()` from within an ISR!

`OS_Replace()` is useful in various situations. For instance, you could have a system initialization task that replaces itself with one of your run-time tasks when all initialization is complete. Or you could replace a large task containing a state machine with independent tasks for each state. `OS_Replace()` can be used wherever multiple tasks need never run at the same time, thus conserving tcb RAM.

In the example below, `TaskCountUp()` runs first. After 250 iterations, it replaces itself with `TaskCountDown()`. `TaskCountDown()` also runs for 250 iterations, but at a faster rate, and replaces itself with `TaskCountUp()` when done. The task priorities can be varied, as shown. This continues indefinitely. Only a single tcb is used.

See Also

`OSCreateTask()`, `OSDestroyTask()`, `OSStop()`

Example

```
void TaskCountUp (void);
void TaskCountDown (void);

_OSLabel(TaskCountUplabel)
_OSLabel(TaskCountDownlabel)

void TaskCountUp( void )
{
    static char i;

    for ( i = 0 ; i <= 250 ; i++ ) {
        PORTB = i;

        OS_Delay(25, TaskCountUplabel);
    }

    OS_Replace(TaskCountDown, 5);
}


void TaskCountDown( void )
{
    static char i;

    for ( i = 250 ; i >= 0 ; i-- ) {
        PORTB = i;

        OS_Delay(5, TaskCountDownlabel);
    }

    OS_Replace(TaskCountUp, 3);
}


main()
{
    …

    OSInit();

    OSCreateTask(TaskCountUp, OSTCBP(1), 4);

    …

    for (;;)
        OSSched();
}
```

OS_SetPrio(): Change the Current Task's Priority and Context-switch

Type:	Macro (invokes `OSSetPrio()`)
Declaration:	`OS_SetPrio ( OStypePrio prio, label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Change the current task's priority. Return to scheduler.
Parameters:	`prio`: the desired (new) priority for the current task. `label`: a unique label.
Returns:	–
Stack Usage:	1

Notes

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest.

Do not call `OS_SetPrio()` from within an ISR!

Tasks can share priorities. Eligible tasks with the same priority will round-robin schedule as long as they are the highest-priority eligible tasks.

The change in priority takes effect when the current task returns to the scheduler.

In the example below, `TaskStartupEtc()` is initially created with a high priority. The first time it runs, it will run at that priority. While running for the first time, it redefines its priority to be a lower one. Each subsequent time it runs, it will run at the lower priority. The task context-switches once at `OS_SetPrio()`, and subsequently at `OS_Yield()`.

See Also

`OSCreateTask()`, `OSGetPrio()`, `OSSetPrio()`, `OSDISABLE_TASK_PRIORITIES`

Example

```
#define MOST_IMPORTANT 0
#define LESS_IMPORTANT 5

main()
{
  …
  /* startup task gets highest priority.       */
  OSCreateTask(TaskStartupEtc,
    OSTCBP(1), MOST_IMPORTANT);
  …
}

/* while starting up this task runs at         */
/*  the highest priority, then it changes      */
/*  its priority to a lower one.               */
void TaskStartupEtc( void )
{
  /* do initialization and other               */
  /*  startup code.                            */
  …

  /* MonitorSystem() will always be            */
  /*  called from this task while              */
  /*  running at a lower priority.             */
  OS_SetPrio(LESS_IMPORTANT, TaskStartupEtc1);

  for (;;) {
    MonitorSystem();

    OS_Yield(TaskStartupEtc2);
  }
}
```

OS_Stop(): Stop the Current Task and Context-switch

Type:	Macro (invokes `OS_Delay()` or `OS_Stop()`)
Declaration:	`OS_Stop ( label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	`OSBYTES_OF_DELAY`, `OSENABLE_STACK_CHECKING`, `OSLOGGING`
Description:	Stop the current task. Return to scheduler.
Parameters:	`label`: a unique label.
Returns:	–
Stack Usage:	1

Notes

A stopped task can only be restarted via `OSStartTask()`.

Do not call `OS_Stop()` from within an ISR!

If delays are enabled via `OSBYTES_OF_DELAYS`, `OS_Stop()` stops the current task via a call to `OSDelay(0)`. Otherwise it calls `OSStop()`. This is done to reduce the code size of your Salvo application.

In the example below, `TaskRunOnce()` is created and started, and will run as soon as it becomes the highest-priority eligible task. It will run only once. In order to make it run again, a call to `OSStartTask(TASK_RUN_ONCE)` is required. Note that `TaskRunOnce()` would also work without the infinite loop, but subsequent calls to `OSStartTask(TASK_RUN_ONCE)` would result in unpredictable behavior because task execution would resume outside of `TaskRunOnce()`.

See Also

`OSStartTask()`, `OSStopTask()`

Example

```
main()
{
  …
  OSCreateTask(TaskRunOnce, TASK_RUN_ONCE_P, 6);
  …
}

void TaskRunOnce( void )
{
  for (;;) {
    /* do one-time things ... */
    …
    OS_Stop(TaskRunOnce1);
  }
}
```

OS_WaitBinSem(): Context-switch and Wait the Current Task on a Binary Semaphore

Type:	Macro (invokes `OSWaitEvent()`)
Declaration:	`OS_WaitBinSem ( OStypeEcbP ecbP, OStypeDelay timeout, label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSENABLE_TIMEOUTS`, `OSLOGGING`
Description:	Wait the current task on a binary semaphore, with a timeout. If the semaphore is 0, return to the scheduler and continue waiting. If the semaphore is 1, reset it to 0 and continue. If the timeout expires before the semaphore becomes 1, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a pointer the binary semaphore's ecb. `timeout`: an integer (>=0) specifying the desired timeout in system ticks. `label`: a unique label.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the binary semaphore indefinitely.

Do not call `OS_WaitBinSem()` from within an ISR!

After a timeout occurs the binary semaphore is undefined.

In the example below for a rocket launching system, a rocket is launched via a binary semaphore `BINSEM_LAUNCH_ROCKET` used as a flag. The semaphore is initialized to zero so that the rocket does not launch on system power-up.[85] Once the rocket is ready and the order has been given to launch (via `OSSignalBinSem()` elsewhere in the code), `TaskLaunchRocket()` starts the rocket on its journey.

[85] That would be undesirable.

Since the rocket cannot be recalled, there is no need to continue running `TaskLaunchRocket()`, and it simply stops itself. Therefore in order to launch a second rocket, the system must be restarted.

See Also

`OSCreateBinSem()`, `OSReadBinSem()`, `OSSignalBinSem()`, `OSTryBinSem()`

Example

```
#define BINSEM_LAUNCH_ROCKET_P OSECBP(2)

…

/* startup code: no clearance given to launch  */
/*  rocket.                                    */
OSCreateBinSem(BINSEM_LAUNCH_ROCKET_P, 0);

…

void TaskLaunchRocket( void )
{
  /* wait here forever until the order is      */
  /* given to launch the rocket.               */
  OS_WaitBinSem(BINSEM_LAUNCH_ROCKET_P,
    OSNO_TIMEOUT, TaskLaunchRocket1);

  /* launch rocket.                            */
  IgniteRocketEngines();
  …

  /* rocket is on its way, therefore task is   */
  /*  no longer needed.                        */
  OS_Stop(TaskLaunchRocket2);
}
```

OS_WaitEFlag(): Context-switch and Wait the Current Task on an Event Flag

Type:	Macro (invokes `OSWaitEvent()`)
Declaration:	`OS_WaitEFlag ( OStypeEcbP ecbP, OStypeEFlag mask, OStypeOption options, OStypeDelay timeout, label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	`OSENABLE_EVENT_FLAGS`, `OSEVENTS`
Affected by:	`OSBYTES_OF_EVENT_FLAGS`, `OSENABLE_STACK_CHECKING`, `OSENABLE_TIMEOUTS`, `OSLOGGING`
Description:	Wait the current task on an event flag, with a timeout. The bits in the event flag specified by the `mask` parameter are tested according to the condition specified by the `options` parameter. If the condition is not satisfied, return to the scheduler and continue waiting. If the condition is satisfied, continue without changing the event flag. If the timeout expires before the condition is satisfied, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a pointer the event flag's ecb. `mask`: a bitmask to apply to the event flag. `options`: `OSANY_BITS`, `OSALL_BITS` or `OSEXACT_BITS`. `timeout`: an integer (>=0) specifying the desired timeout in system ticks. `label`: a unique label.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the event flag indefinitely.

Do not call `OS_WaitEFlag()` from within an ISR!

After a timeout occurs the event flag is undefined.

Salvo's event flag bits are "active high", i.e. an event is said to have occurred when its corresponding bit in the event flag is set to 1. The event has not occurred if the bit is cleared to 0.

When specifying `OSANY_BITS`, `OS_WaitEFlag()` checks if any of the corresponding `mask` parameter's bits in the event flag are set to 1, and if so, the task continues. With `OSALL_BITS`, all of the corresponding `mask` parameter's bits must be set to 1 for the task to continue. With `OSEXACT_BITS`, the event flag must match the `mask` parameter exactly for the task to continue.

In contrast to Salvo's other event services, successfully waiting an event flag *does not automatically reset the bits in the event flag* that resulted in the condition being satisfied. You must explicitly clear event flag bits via `OSClrEFlag()`. Failing to clear the appropriate event flag bits will cause unpredictable results – generally the task will fail to yield back to the scheduler.

In the example below for a secure access system with a power-assisted door, three separate interlocks must be deactivated before the door can be opened by `TaskOpenDoor()`. The three least significant bits of an eight-bit event flag are used to signify that the bottom, side and top interlocks have been deactivated by `TaskReleaseBottomLock()`, etc. Bits three and four in the event flag signify whether the door is fully open or fully closed and are maintained by `TaskCheckDoor()`. When the door is fully open, it's safe to re-activate (release) the door locks so that when it closes it's automatically locked shut.

The remaining three bits in the eight-bit event flag can be used for other purposes entirely independent of the interlock mechanism.

See Also

`OSCreateEFlag()`, `OSClrEFlag()`, `OSReadEFlag()`, `OSSetEFlag()`

Example

```
#define DOOR_EFLAG_P   OSECBP(1)
#define BOTTOM         0x01
#define SIDE           0x02
#define TOP            0x04
#define OPEN           0x08
#define CLOSED         0x10

void TaskReleaseBottomLock( void )
{
  for (;;) {
    /* wait for request to release bottom lock.*/
```

```
    …
    /* release bottom door lock.                */
    ReleaseBottomLock();

    /* tell TaskOpenDoor() about it.            */
    OSSetEFlag(DOOR_EFLAG_P, BOTTOM);

    /* verify that door is fully opened by      */
    /*  by waiting for the signal.              */
    OS_WaitEFlag(DOOR_EFLAG_P, OPEN, OSANY_BITS,
      OSNO_TIMEOUT, TaskReleaseBottomLock1);

    /* re-engage bottom door lock. When door    */
    /*  closes it will remain locked.           */
    OSClrEFlag(DOOR_EFLAG_P, BOTTOM);
    EngageBottomLock();

    /* remain inactive until the door closes.   */
    OS_WaitEFlag(DOOR_EFLAG_P, CLOSED, OSANY_BITS,
      OSNO_TIMEOUT, TaskReleaseBottomLock2);
  }
}

void TaskReleaseSideLock( void )
{
  for (;;) {
    …
    ReleaseSideLock();
    OSSetEFlag(DOOR_EFLAG_P, SIDE);
    OS_WaitEFlag(DOOR_EFLAG_P, OPEN, OSANY_BITS,
      OSNO_TIMEOUT, TaskReleaseSideLock1);
    OSClrEFlag(DOOR_EFLAG_P, SIDE);
    EngageSideLock();
    OS_WaitEFlag(DOOR_EFLAG_P, CLOSED, OSANY_BITS,
      OSNO_TIMEOUT, TaskReleaseSideLock2);
  }
}

void TaskReleaseTopLock( void )
{
  for (;;) {
    …
    ReleaseTopLock();
    OSSetEFlag(DOOR_EFLAG_P, TOP);
    OS_WaitEFlag(DOOR_EFLAG_P, OPEN, OSANY_BITS,
      OSNO_TIMEOUT, TaskReleaseTopLock1);
    OSClrEFlag(DOOR_EFLAG_P, TOP);
    EngageTopLock();
    OS_WaitEFlag(DOOR_EFLAG_P, CLOSED, OSANY_BITS,
      OSNO_TIMEOUT, TaskReleaseTopLock2);
  }
}

void TaskOpenTheDoor( void )
{
  /* door is initially closed.                  */
  OSCreateEFlag(DOOR_EFLAG_P, CLOSED );
```

```
  for (;;) {
    /* wait forever for all interlocks to be   */
    /*  released.                              */
    OS_WaitEFlag(DOOR_EFLAG_P,
      TOP | BOTTOM | SIDE, OSALL_BITS,
      OSNO_TIMEOUT, TaskOpenTheDoor1);

    /* all locks are released – open door.     */
    OpenDoor();

    /* wait for the door to close again before */
    /*  repeating the cycle.                   */
    OS_WaitEFlag(DOOR_EFLAG_P, CLOSED, OSANY_BITS,
      OSNO_TIMEOUT, TaskOpenTheDoor2);
  }
}

void TaskCheckDoor( void )
{
  for (;;) {
    /* check sensors every 1s.                 */
    OS_Delay(100, TaskCheckDoor1);

    /* if open door has closed contact on its  */
    /*  sensor, then door must be open!        */
    if ( DoorFullyOpen() )
      OSSetEFlag(DOOR_EFLAG_P, OPEN);
    else
      OSClrEFlag(DOOR_EFLAG_P, OPEN);

    /* similarly, if closed door has closed    */
    /*  contact on its sensor, then it must be */
    /*  closed!                                */
    if ( DoorFullyClosed() )
      OSSetEFlag(DOOR_EFLAG_P, CLOSED);
    else
      OSClrEFlag(DOOR_EFLAG_P, CLOSED);
  }
}
```

OS_WaitMsg(): Context-switch and Wait the Current Task on a Message

Type:	Macro (invokes `OSWaitEvent()`)
Declaration:	`OS_WaitMsg ( OStypeEcbP ecbP, OStypeMsg msgP, OStypeDelay timeout, label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	`OSENABLE_MESSAGES`, `OSEVENTS`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSENABLE_TIMEOUTS`, `OSLOGGING`
Description:	Wait the current task on a message, with a timeout. If the message is available, make `msgP` point to it, and continue. If it's not available, return to the scheduler and continue waiting. If the timeout expires before the message becomes available, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a to the message's ecb. `msgP`: a pointer to a message `timeout`: an integer (>=0) specifying the desired timeout in system ticks. `label`: a unique label.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the message indefinitely.

Do not call `OS_WaitMsg()` from within an ISR!

After a timeout occurs the message pointer is invalid.

In the example below, `TaskRcvKeys()` waits forever for the message `MSG_KEY_PRESSED`. No processing power is allocated to `TaskRcvKeys()` while it is waiting. Once the message arrives, its contents (the key pressed) are copied to a local variable and appropriate action is taken. Note that correct casting and dereferencing of the pointer msgP are required in order to extract the contents of the message correctly. After `TaskRcvKeys()` acts on the key pressed, it resumes waiting for the message.

See Also `OSCreateMsg()`, `OSReadMsg()`, `OSSignalMsg()`, `OSTryMsg()`

Example

```
void TaskRcvKeys( void )
{
  static char key;
  static OStypeMsgP msgP;

  for (;;) {
    /* Wait forever for a new key.              */
    OS_WaitMsg(MSG_KEY_PRESSED_P,
      &msgP, OSNO_TIMEOUT, TaskRcvKeys1);

    /* User pressed a key! – get it.            */
    key = *(char *) msgP;

    /* Act on key pressed.                      */
    switch ( tolower(key) ) {
      case KEY_MEM:
        …
    }
  }
}
```

OS_WaitMsgQ(): Context-switch and Wait the Current Task on a Message Queue

Type:	Macro (invokes `OSWaitEvent()`)
Declaration:	`OS_WaitMsgQ ( OStypeEcbP ecbP, OStypeMsg msgP, OStypeDelay timeout, label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`, `OSEVENTS`
Affected by:	`OSLOGGING`, `OSENABLE_STACK_CHECKING`
Description:	Wait the current task on a message queue, with a timeout. If the message queue contains a message, make `msgP` point to it, and continue. If it's empty, return to the scheduler and continue waiting. If the timeout expires before a message is added to the message queue, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a pointer to the message queue's ecb. `msgP`: a pointer to a message. `timeout`: an integer (>=0) specifying the desired timeout in system ticks. `label`: a unique label.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the message queue indefinitely.

Do not call `OS_WaitMsgQ()` from within an ISR!

After a timeout occurs the message pointer is invalid.

In the example below, a `TaskRcvInt()` waits forever a message queue containing messages to objects of type `int`. When a message arrives, the `TaskRcvInt()` extracts the message from the message queue and prints a message. The task continues printing messages until the message queue is empty, whereupon the task a context switch occurs.

See Also

`OSCreateMsgQ()`, `OSReadMsgQ()`, `OSSignalMsgQ()`, `OSTryMsgQ()`

Example

```
void TaskRcvInt( void )
{
  static int myNum;
  static OStypeMsgP msgP;

  for (;;) {
    /* Wait forever for a message.            */
    OS_WaitMsgQ(MSGQ1, &msgP, OSNO_TIMEOUT,
      TaskRcvInt1);

    /* A message has arrived – get it.        */
    myNum = *(int *) msgP;

    printf("The number was %d. \n", myNum);
  }
}
```

OS_WaitSem(): Context-switch and Wait the Current Task on a Semaphore

Type:	Macro (invokes `OSWaitEvent()`)
Declaration:	`OS_WaitSem ( OStypeEcbP ecbP, OStypeDelay timeout, label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	`OSENABLE_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSENABLE_TIMEOUTS`, `OSLOGGING`
Description:	Wait the current task on a semaphore, with a timeout. If the semaphore is 0, return to the scheduler and continue waiting. If the semaphore is non-zero, decrement the semaphore and continue. If the timeout expires before the semaphore becomes non-zero, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a pointer to the semaphore's ecb. `timeout`: an integer (>=0) specifying the desired timeout in system ticks. `label`: a unique label.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the semaphore indefinitely.

Do not call `OS_WaitSem()` from within an ISR!

After a timeout occurs the semaphore is undefined.

In the example below, `TaskRcvRsp()` removes incoming characters from a receive buffer one at a time and processes them. `SEM_RX_BUFF` always indicates how many characters are present in `rxBuff[]`, and is signaled by another task which puts the characters into `rxBuff[]` one-by-one. `TaskRcvRsp()` runs as long as there are characters present in `rxBuff[]` – when is empty, `TaskRcvRsp()` waits. By using a semaphore for inter-task communications there's no need to poll for the existence of characters in the buffer, and hence overall performance is improved.

See Also `OSCreateSem()`, `OSReadSem()`, `OSSignalSem()`, `OSTrySem()`

Example

```
void TaskRcvRsp( void )
{
  static char rcChar;

  for (;;) {
    /* wait until there are response chars     */
    /*  waiting ... (TaskRx() signals us when  */
    /*  there are).                            */
    OS_WaitSem(SEM_RX_RBUFF_P, OSNO_TIMEOUT,
      TaskRcvRsp1);

    /* then deal with them.                    */
    /* get the next char from the buffer       */
    rcChar = rxBuff[rxHead];
    rxHead++;
    if ( rxHead >= SIZEOF_RX_BUFF )
      rxHead = 0;
    rxCount--;

    /* alphanumeric characters are the _only_  */
    /*  chars (other than reserved ones) we    */
    /*  expect to see in the incoming rcChar.  */
    if ( isalnum(rcChar) || ( rcChar == '-' ) ) {
      …
    }
    else {
      …
    }
  }
}
```

OS_Yield(): Context-switch

Type:	Macro
Declaration:	`OS_Yield ( label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	–
Description:	Return to scheduler.
Parameters:	`label`: a unique label.
Returns:	–
Stack Usage:	1 or 2, depending on compiler and target.

Notes

`OS_Yield()` causes an immediate, unconditional return to the scheduler.

Do not call `OS_Yield()` from within an ISR!

In the example below, `TaskUnimportant()` is assigned a low priority and runs only when no other higher-priority tasks are eligible to run. Each time it runs, it increments a counter by 1.

Example

```
unsigned long int unimportantCounter = 0;

int main ( void )
{

  OSCreateTask(TaskUnimportant,
    TASK_UNIMPORTANT_P, 14);

  …
}

void TaskUnimportant( void )
{
  for (;;) {
    unimportantCounter++;

    OS_Yield(TaskUnimportant1);
  }
}
```

OSClrEFlag(): Clear Event Flag Bit(s)

Type:	Function
Prototype:	`OStypeErr OSClrEFlag ( OStypeEcbP ecbP, OStypeEFlag mask );`
Callable from:	Anywhere
Contained in:	`eflag.c`, `event.c`
Enabled by:	`OSENABLE_EVENT_FLAGS`, `OSEVENTS`
Affected by:	`OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Clear bits in an event flag. No task will be made eligible by this operation.
Parameters:	`ecbP`: a pointer to the event flag's ecb. `mask`: mask of bits to be cleared.
Returns:	`OSERR_BAD_P` if event flag pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not an event flag. `OSERR_EVENT_CB_UNINIT` if event flag's control block is uninitialized. `OSERR_EVENT_FULL` if event flag doesn't change. `OSNOERR` if event flag bits are successfully cleared.
Stack Usage:	1

Notes

No tasks are made eligible by clearing bits in an event flag.

This service is typically used immediately after successfully waiting an event flag, since the bits in question are not automatically cleared by `OS_WaitEFlag()`.

In the example below, a task is configured to run only when two particular bits in an event flag are set. It then clears one of them and returns to the waiting state. It will run again when and only when both bits are set.

See Also

`OS_WaitEFlag()`, `OSCreateEFlag()`, `OSReadEFlag()`, `OSSetEFlag()`

Example

```
#define EFLAG1_P OSECBP(2)
…
void TaskC( void )
{
  for (;;) {
    /* wait forever for both bits to be set    */
    OS_WaitEFlag(EFLAG1_P, 0x0C, OSALL_BITS,
      OSNO_TIMEOUT, TaskC1);

    /* clear the upper bit, leave the lower    */
    /*  one alone.                             */
    OSClrEFlag(EFLAG1_P, 0x08);

    …
  }
}
```

OSCreateBinSem(): Create a Binary Semaphore

Type:	Function
Prototype:	`OStypeErr OSCreateBinSem ( OStypeEcbP ecbP, OStypeBinSem binSem );`
Callable from:	Anywhere
Contained in:	`binsem.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create a binary semaphore with the initial value specified.
Parameters:	`ecbP`: a pointer to the binary semaphore's ecb. `binSem`: the binary semaphore's initial value (0 or 1) .
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating a binary semaphore assigns an event control block (ecb) to the semaphore.

A newly-created binary semaphore has no tasks waiting for it.

Signaling or waiting a binary semaphore before it has been created will result in an error if `OSUSE_EVENT_TYPES` is `TRUE`.

You can also implement binary semaphores via messages – see `OSCreateMsg()`.

In the example below, a binary semaphore is used to control access to a shared resource, an I/O port. The port is initially available for use, so the semaphore is initialized to 1.

See Also

`OS_WaitBinSem()`, `OSReadBinSem()`, `OSSignalBinSem()`, `OSTryBinSem()`

Example

```
/* PORTB is a general-purpose I/O port.        */
#define BINSEM_PORTB_P OSECBP(6)
…
/* PORTB is initially available to task that   */
/*  wants to use it.                           */
OSCreateBinSem(BINSEM_PORTB_P, 1);
…
```

OSCreateEFlag(): Create an Event Flag

Type:	Function
Prototype:	`OStypeErr OSCreateEFlag ( OStypeEcbP ecbP, OStypeEfcbP efcbP, OStypeEFlag eFlag );`
Callable from:	Anywhere
Contained in:	`eflag.c`
Enabled by:	`OSENABLE_EVENT_FLAGS`, `OSEVENTS`
Affected by:	`OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create an event flag with the initial value specified.
Parameters:	`ecbP`: a pointer to the event flag's ecb. `efcbP`: a pointer to the event flag's efcb. `eFlag`: the event flag's initial value.
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating an event flag assigns an event control block (ecb) and an event flag control block (efcb) to the event flag.

A newly-created event flag has no tasks waiting for it.

Signaling or waiting an event flag before it has been created will result in an error if `OSUSE_EVENT_TYPES` is `TRUE`.

Event flags can be 8, 16 or 32 bits, depending on `OSBYTES_OF_EVENT_FLAGS`. `OSCreateEFlag()` stores the value of the event flag in the event flag's pre-existing event flag control block (efcb) of type `OSgltypeEfcb`.

In the example below, an 8-bit event flag is used to signify the occurrence of keypresses from an 8-key machine control keypad. Each bit maps to a single key. The event flag is initialized to all `0`'s to indicate that no keypresses have occurred. `OSBYTES_OF_EVENT_FLAGS` is set to 1 in this example's `salvocfg.h`.

See Also

`OS_WaitEFlag()`, `OSReadEFlag()`, `OSSignalEFlag()`, `OSTryEFlag()`

Example

```
/* event flag #3.                                   */
#define EFLAG_KEYS_P  OSECBP(3)
…
/* allocate RAM for the ecfb.                       */
OsgltypeEfcb eFlagKeys;
…
/* Initially no keys have been pressed.             */
OSCreateEFlag(EFLAG_KEYS_P, &eFlagKeys, 0x00);
…
```

OSCreateMsg(): Create a Message

Type:	Function
Prototype:	`OStypeErr OSCreateMsg ( OStypeEcbP ecbP, OStypeMsgP msgP );`
Callable from:	Anywhere
Contained in:	`msg.c`
Enabled by:	`OSENABLE_MESSAGE`, `OSEVENTS`
Affected by:	`OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create a message with the initial value specified.
Parameters:	`ecbP`: a pointer to the message's ecb. `msgP`: a pointer to a message.
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating a message assigns an event control block (ecb) to the message. A newly-created message has no tasks waiting for it. Messages are passed via pointer so that a message can point to anything.

Signaling or waiting a message before it has been created will result in an error if `OSUSE_EVENT_TYPES` is `TRUE`.

Binary semaphores and resource locking can be implemented via messages using the values `(OStypeMsgP) 0` and `(OStypeMsgP) 1` for the messages.

In the example below, a message is created to pass the key pressed (which is detected by the task `TaskReadKey()`) to the task `TaskHandleKey()`, which acts on the keypress. The message is initialized to zero because no keypress is initially detected. If, due to task priorities and timing, `TaskReadKey()` signals a new message before `TaskHandleKey()` reads the existing message, the new key will be lost.

See Also

`OS_WaitMsg()`, `OSReadMsg()`, `OSSignalMsg()`, `OSTryMsg()`

Example

```
/* pass key via a message. */
```

```
#define MSG_KEY_PRESSED_P OSECBP(4)
…
/* this task reads key presses from a keypad   */
/*  and sends them to TaskHandleKey via a      */
/*  message.                                   */
void TaskReadKey( void )
{
  static char key;     /* holds key pressed    */

  /* initially no key has been pressed.        */
  OSCreateMsg(MSG_KEY_PRESSED_P, (OStypeMsgP) 0);

  for (;;) {
    if ( kbhit() ) {
      key = getch();

      /* do debouncing, key-repeat, etc.       */

      /* send new key via message.             */
      OSSignalMsg(MSG_KEY_PRESSED_P,
        (OStypeMsgP) &key);
    }

    /* wait 10msec, then test for keypress     */
    /*  again.                                 */
    OS_Delay(TEN_MSEC, TaskReadKey1);
  }
}

/* this task acts upon keypresses.             */
void TaskHandleKey( void )
{
  static char key;         /* holds new key    */
  static OStypeMsgP msgP;  /* get msg via ptr  */

  for (;;) {
    /* do nothing until a key is pressed.      */
    OS_WaitMsg(MSG_KEY_PRESSED_P, &msgP,
      OSNO_TIMEOUT, TaskHandleKey1);

    /* then get the new key and act on it.     */
    key = *(char *)msgP;
    switch ( tolower(key) ) {
      case KEY_UP:
        MoveUp();
        break;
      …
    }
  }
}
```

OSCreateMsgQ(): Create a Message Queue

Type:	Function
Prototype:	`OStypeErr OSCreateMsgQ ( OStypeEcbP ecbP, OStypeMqcbP mqcbP, OStypeMsgQPP msgPP, OStypeMsgQSize size );`
Callable from:	Anywhere
Contained in:	`msgq.c`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`, `OSEVENTS`
Affected by:	`OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create an empty message queue.
Parameters:	`ecbP`: a pointer to the message queue's ecb. `mqcbP`: a pointer to the message queue's message queue control block. `msgPP`: a pointer to the buffer that will hold the message queue's message pointers. `size`: the number of messages (`0` < `size` < `256`) that the message queue can hold.
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating a message queue assigns an event control block (ecb) to the message.

Each message queue has a *message queue control block* (mqcb) associated with it. Salvo message queue services use mqcbs to manage the insertion and removal of messages into and out of each message queue. You must allocate memory for mqcbs using the `OSMESSAGE_QUEUES` configuration option. You must associate a unique mqcb with each message queue using a *message queue control block pointer*. These range from `OSMQCBP(1)` to `OSMQCBP(OSMESSAGE_QUEUES)`. A newly-created message queue contains no messages.

A message queue[86] holds its message pointers[87] within a circular buffer. You must declare this buffer in your source code as a simple array, and give `OSCreateMsgQ()` a handle to it via the `msgPP`

[86] Of type `OSgltypeMsgQP`.

[87] Of type `OStypeMsgP`.

parameter. The buffer must hold `size` message pointers. `OSCreateMsgQ()` does not have any effect on the contents of the buffer.

In the example below, a 7-element and a 16-element message queue are created with the buffers `MsgQBuff1[]` and `MsgQBuff2[]`, respectively. The message queue control block IDs are 1 and 2, since memory was allocated for two message queues via `OSMESSAGE_QUEUES` in `salvocfg.h`.

For this example `salvocfg.h` contains:

```
#define OSEVENTS         5
#define OSMESSAGE_QUEUES 2
```

In this example, all of the `OSLOC_XYZ` configuration options are at their default values. By using `OSLOC_MSGQ` and `OSLOC_MQCB` you can relocate the buffers and the mqcbs, respectively, into RAM banks other than the default banks.

See Also

`OS_WaitMsgQ()`, `OSReadMsgQ()`, `OSSignalMsgQ()`, `OSTryMsgQ()`, `OSLOC_MSGQ`, `OSLOC_MQCB`

Example

```
/* use #defines for legibility                */
#define SEM1_P       OSECBP(1)
#define SEM2_P       OSECBP(2)
#define BINSEM1_P    OSECBP(3)
#define MSGQ1_P      OSECBP(4)
#define MSGQ2_P      OSECBP(5)
#define MQCB1_P      OSMQCBP(1)
#define MQCB2_P      OSMQCBP(2)
#define SIZEOF_MSGQ1 7
#define SIZEOF_MSGQ2 16

/* allocate memory for buffers                */
OSgltypeMsgQP MsgQBuff1[SIZEOF_MSGQ1];
OSgltypeMsgQP MsgQBuff2[SIZEOF_MSGQ2];

/* create message queues from existing        */
/*  buffers and mqcbs.                        */
OSCreateMsgQ(MSGQ1_P, MQCBP1_P, MsgQBuff1,
  SIZEOF_MSGQ1);
OSCreateMsgQ(MSGQ2_P, MQCBP2_P, MsgQBuff2,
  SIZEOF_MSGQ2);
```

OSCreateSem(): Create a Semaphore

Type:	Function
Prototype:	`OStypeErr OSCreateSem ( OStypeEcbP ecbP, OStypeSem sem );`
Callable from:	Anywhere
Contained in:	`sem.c`
Enabled by:	`OSENABLE_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSBIG_SEMAPHORES`, `OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create a counting semaphore with the initial value specified.
Parameters:	`ecbP`: a pointer to the semaphore's ecb. `sem`: the semaphore's initial value.
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating a semaphore assigns an event control block (ecb) to the semaphore.

A newly-created semaphore has no tasks waiting for it.

Signaling or waiting a semaphore before it has been created will result in an error if `OSUSE_EVENT_TYPES` is `TRUE`.

In the example below, a counting semaphore is created to mark how much space is available in a transmit ring buffer. The buffer is initially empty, so the semaphore is initialized to the size of the buffer.

See Also

`OS_WaitSem()`, `OSReadSem()`, `OSSignalSem()`, `OSTrySem()`

Example

```
/* Ring buffer is used to receive characters.  */
#define SEM_TX_RBUFF_P OSECBP(3)

…

/* initialize semaphore (ring buffer is        */
/*  empty).                                    */
OSCreateSem(SEM_TX_RBUFF_P, 16);

…
```

OSCreateTask(): Create and Start a Task

Type:	Function
Prototype:	`OStypeErr OSCreateTask ( OStypeTFP tFP, OStypeTcbP tcbP, OStypePrio prio );`
Callable from:	Background only
Contained in:	`inittask.c`
Enabled by:	–
Affected by:	`OSLOGGING`, `OSENABLE_STACK_CHECKING`
Description:	Create a task with the specified start address, tcb pointer and priority. Starts the task unless overridden by the user in the `prio` parameter.
Parameters:	`tFP`: a pointer to the task's start address. This is also the task's function prototype name. `tcbP`: a pointer to the task's tcb. `prio`: the desired priority for the task. If OR'd with `OSDONT_START_TASK`, the task will not be started.
Returns:	`OSNOERR` if task is successfully created. `OSERR_BAD_P` if the specified tcb pointer is invalid (i.e. out-of-range).
Stack Usage:	3

Notes

Creating a task assigns a task control block (tcb) to the task.

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest. If the specified task priority is out-of-range, the task will still be created, but with the lowest possible priority.

Tasks created via `OSCreateTask()` are automatically started, i.e. they are in the eligible state.

If you prefer to create the task now and explicitly start it later, OR `OSCreateTask()`'s `prio` parameter with `OSDONT_START_TASK`. Then use `OSStartTask()` to start the task at a later time.

If task priorities are disabled via `OSDISABLE_TASK_PRIORITIES`, `OSCreateTask()`'s third argument (`prio`) is used only with `OSDONT_START_TASK`, and the priority value is disregarded.

Caution `OSCreateTask()` overwrites the task control block specified via the `tcbP` parameter, i.e. it overwrites the tcb. When calling `OSCreateTask()` after task scheduling has started via `OSSched()`, extreme caution must be used to avoid overwriting an existing eligible, running, delayed, waiting or stopped task.

In the example below, a single task is created from the function `TaskDoNothing()` by assigning it a tcb pointer of `TASK1_P`, and a priority of 7.

See Also

`OSStartTask()`, `OSStopTask()`

Example

```
#define TASK1_P OSTCBP(1)/* taskIDs start at 0 */

/* this task does nothing but run, context-    */
/*  switch, run, context-switch, etc.          */
void TaskDoNothing( void )
{
  for (;;)
    OS_Yield(TaskDoNothing1);
}

/* create a single task and run it (over and   */
/*  over).                                     */
void main( void )
{
  …
  /* initialize Salvo. */
  OSInit();

  /* create a task to do nothing but context-  */
  /*  switch. Tcb pointer is 0, priority is 7  */
  /*  (middle). A call to OSSTartTask() is not */
  /*  required …                               */
  OSCreateTask(TaskDoNothing, TASK1_P, 7);

  …
  /* start multitasking.                       */
  for (;;)
    OSSched();
}
```

OSDestroyTask(): Destroy a Task

Type:	Function
Prototype:	`OStypeErr OSDestroyTask ( OStypeTcbP tcbP );`
Callable from:	Task or Background
Contained in:	`task.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Destroy the specified task.
Parameters:	`tcbP`: a pointer to the task's tcb.
Returns:	`OSNOERR` if specified task was successfully destroyed. `OSERR` if unable to destroy the specified task.
Stack Usage:	3

Notes

`OSDestroyTask()` can destroy tasks that are stopped (`OSTA_STOPPED`), delayed (`OSTA_DELAYED`), eligible (`OSTA_ELIGIBLE`) or have timed out (`OSTA_TIMEDOUT`). It cannot destroy tasks in other states.

The destroyed task's tcb is re-initialized.

In the example below, `TaskMain()` has a relatively high priority of 3. When it runs, it creates another, lower-priorty task, `TaskWarmUp()`. During the next thirty seconds, `TaskWarmUp()` runs whenever it is the highest-priority eligible task. Then `TaskMain()` destroys `TaskWarmUp()`. Thereafter, `OSCreateTask()` can be used to create another task in `TaskWarmUp()`'s place, using the same tcb pointer.

See Also

`OSCreateTask()`, `OS_Destroy()`

Example

```
OSCreateTask(TaskMain, TASKMAIN, 3);
…
void TaskMain ( void )
{
  OSCreateTask(TaskWarmUp, TASKWARMUP_P, 7);

  for (;;) {
    OS_Delay(THIRTY_SEC, TaskMain1);
    OSDestroyTask(TASKWARMUP_P);
    …
  }
}
```

OSGetPrio(): Return the Current Task's Priority

Type:	Macro (invokes `OSGetPrioTask()`)
Prototype:	`OStypePrio OSGetPrio ( );`
Callable from:	Task only
Contained in:	`prio2.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Return the priority of the current (running) task.
Parameters:	–
Returns:	–
Stack Usage:	1

Notes

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest.

In the example below, `TaskB()` lowers its priority each time it runs, until it reaches the lowest allowed priority and remains there.

See Also

`OS_SetPrio()`, `OSGetPrioTask()`, `OSSetPrio()`, `OSSetPrioTask()`, `OSDISABLE_TASK_PRIORITIES`

Example

```
void TaskB( void )
{
  OStypePrio prio;


  for (;;) {
    …
    prio-- = OSGetPrio();
    OS_SetPrio(prio, TaskB1);
    }
  }
}
```

OSGetPrioTask(): Return the Specified Task's Priority

Type:	Function
Prototype:	`OStypePrio OSGetPrioTask ( OStypeTcbP tcbP );`
Callable from:	Task or Background
Contained in:	`prio2.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Return the priority of the specified task.
Parameters:	`tcbP`: a pointer to the task's tcb.
Returns:	–
Stack Usage:	1

Notes

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest.

In the example below, `DispTaskPrio()` displays the priority of the specified task.

See Also

`OS_SetPrio()`, `OSGetPrio()`, `OSSetPrio()`, `OSSetPrioTask()`, `OSDISABLE_TASK_PRIORITIES`

Example

```
#define TASKE_P  OSTCBP(5)
…
void DispTaskPrio( OStypeTcbP tcbP )
{
  printf("Task %d has priority %d.\n",
   OStID(tcbP), OSGetPrioTask(tcbP));
}
```

OSGetState(): Return the Current Task's State

Type:	Macro (invokes `OSGetStateTask()`)
Prototype:	`OStypeState OSGetState ( );`
Callable from:	Task only
Contained in:	`task.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Return the state of the current (running) task.
Parameters:	–
Returns:	Task state.
Stack Usage:	1

Notes

The current task's state is always `OSTA_RUNNING`. This service is included for completeness.

In the example below, `TaskG()` verifies that it is in fact running.

See Also

`OSGetStateTask()`

Example

```
void TaskC( void )
{
  for (;;) {
    if ( OSGetState() != OSTA_RUNNING )
      printf("Houston, we have a problem.\n");
  }
}
```

OSGetStateTask(): Return the Specified Task's State

Type:	Function
Prototype:	`OStypeState OSGetState ( OStypeTcbP tcbP );`
Callable from:	Task or Background
Contained in:	`task.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Return the state of the specified task.
Parameters:	–
Returns:	Task state.
Stack Usage:	1

Notes

A task may be in one of the following states:

`OSTA_DESTROYED`	destroyed / uninitialized
`OSTA_STOPPED`	stopped
`OSTA_DELAYED`	delayed
`OSTA_WAITING`	waiting on an event
`OSTA_WAITING_TO`	waiting on an event, with a timeout
`OSTA_ELIGIBLE`	eligible to run
`OSTA_TIMEDOUT`	timed out
`OSTA_RUNNING`	running

In the example below, mainline code verifies that a particular task has indeed been stopped.

See Also

`OSGetState()`

Example

```
#define TASKC_P OSTCBP(3)
…
if ( OSGetStateTask(TASKC_P) != OSTA_STOPPED )
  /* something's wrong with TaskC().          */
…
```

OSGetTicks(): Return the System Timer

Type:	Function
Prototype:	`OStypeTick OSGetTicks ( void );`
Callable from:	Anywhere
Contained in:	`tick.c`
Enabled by:	`OSBYTES_OF_TICKS`
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Obtain the current value of the system timer (in ticks).
Parameters:	–
Returns:	Current system timer in ticks.
Stack Usage:	1

Notes

The system timer is initialized to 0 via `OSInit()`.

In the example below, the current value of the system timer is stored in a variable.

See Also

`OSSetTicks()`

Example

```
…

OStypeTick ticksNow;

…

/* obtain current value of system ticks.       */
ticksNow = OSGetTicks();
…
```

On certain targets it may be advantageous to read the current system ticks (`OStimerTicks`) directly instead of through `OSGetTicks()`. Possible scenarios include substantial function call overhead and/or no need to manage interrupts.[88] In the example below, the current value of the system timer is stored in a variable by accessing `OStimerTicks` directly.

```
…

OStypeTick ticksNow;

…

/* obtain current value of system ticks.       */
OSDi();
ticksNow = OStimerTicks;
OSEi();
…
```

[88] Both of these conditions occur on the baseline PICmicro devices, e.g. PIC12C509.

OSGetTS(): Return the Current Task's Timestamp

Type:	Macro (invokes `OSGetTSTask()`)
Prototype:	`OStypeTS OSGetTS ( void );`
Callable from:	Task only
Contained in:	`delay3.c`
Enabled by:	`OSBYTES_OF_TICKS`
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Obtain the value of the current task's timestamp (in ticks).
Parameters:	–
Returns:	Current task's timestamp in ticks.
Stack Usage:	1

Notes

When a task is created, its timestamp is initialized to an `OStypeTS`-sized version of the system timer ticks, i.e. `(OStypeTS) OStimerTicks`.

In the example below, the current task's timestamp is displayed whenever it times out.

See `OS_DelayTS()` for more information on timestamps.

See Also

`OS_DelayTS()`, `OSSetTS()`, `OSSyncTS()`

Example

```
void Task( void )
{
  while ( TRUE ) {
    OS_Delay(7, label);[89]

    printf("Task %d timed out at %d\n",
      OStID(OScTcbP), OSGetTS());

    …
  }
}
```

[89] The timestamp is redefined whenever a delay expires, whether through `OS_Delay()` or `OS_DelayTS()`.

OSInit(): Prepare for Multitasking

Type:	Function
Prototype:	`void OSInit ( void );`
Callable from:	Background only
Contained in:	`init.c`
Enabled by:	–
Affected by:	`OSBYTES_OF_DELAYS`, `OSCLEAR_GLOBALS`, `OSENABLE_STACK_CHECKING`, `OSEVENTS`, `OSLOGGING`, `OSTASKS`
Description:	Initialize Salvo's pointers, counters, etc.
Parameters:	–
Returns:	–
Stack Usage:	2

Notes

`OSInit()` must be called first, before any other Salvo functions.

The executable code size of `OSInit()` can be minimized by setting `OSCLEAR_GLOBALS` to `FALSE`. Do this only if you are certain that your compiler initializes all global variables to 0 at runtime, and you do not call `OSInit()` more than once in your application.

`OSInit()` does not initialize tcbs or ecbs – this is done on a per-tcb and per-ecb basis when tasks and events are created, respectively.

In the example below, `OSInit()` is called before any other Salvo calls.

Example

```
void main( void )
{
  …
  /* initialize Salvo.                          */
  OSInit();
  …
  /* start multitasking.                        */
  for (;;)
    OSSched();
}
```

OSMsgQCount(): Return Number of Messages in Message Queue

Type:	Function
Prototype:	`OStypeMsgQSize OSMsgQCount ( OStypeTcbP ecbP );`
Callable from:	Anywhere
Contained in:	`msgq4.c`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`
Affected by:	`OSCALL_OSMSGQCOUNT`
Description:	Check whether the specified message queue has room for additional message(s).
Parameters:	`ecbP`: a pointer to the message queue's ecb.
Returns:	Number of messages in message queue, i.e. returns 0 if message queue is empty.
Stack Usage:	1

Notes

`OSMsgQCount()` can be used to obtain the current status of the message queue. `OSMsgQCount()` returns the `count` record in the message queue's message queue control block (mqcb) – therefore it's very fast.

No error checking is performed on the `ecbP` parameter. Calling `OSMsgQCount()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a message queue, will return an erroneous result.

In the example below, `OSMsgQCount()` is used to obtain the number of messages in a message queue, and the space available for new messages. When using `OSMsgQCount()` to calculate available space in a message queue, it must be subtracted from the size parameter originally used to create the message queue.

See Also

`OS_WaitMsgQ()`, `OSCreateMsgQ()`, `OSMsgQEmpty()`, `OSReadMsgQ()`, `OSSignalMsgQ()`, `OSTryMsgQ()`

Example

```
#define MSGQ1_P OSECBP(1)

printf("msgQ contains %d messages\n",
  OSMsgQCount(MSGQ1_P));
printf("msgQ has room for %d messages\n",
  SIZEOF_MSGQ1 - OSMsgQCount(MSGQ1_P));
```

OSMsgQEmpty(): Check for Available Space in Message Queue

Type:	Function
Prototype:	`OStypeMsgQSize OSMsgQEmpty ( OStypeTcbP ecbP );`
Callable from:	Anywhere
Contained in:	`msgq3.c`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`
Affected by:	`OSCALL_OSMSGQEMPTY`
Description:	Check whether the specified message queue has room for additional message(s).
Parameters:	`ecbP`: a pointer to the message queue's ecb.
Returns:	Number of available (empty) spots in message queue, i.e. returns 0 (FALSE) if message queue is full.
Stack Usage:	1

Notes

Each message queue can contain up to a maximum number of messages. If messages are added to the message queue (via `OSSignalMsgQ()`) faster than they are removed (via `OS_WaitMsgQ()`), the queue will eventually fill up. `OSMsgQEmpty()` can be used to obtain the current status of the message queue without signaling the message queue.

No error checking is performed on the `ecbP` parameter. Calling `OSMsgQEmpty()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a message queue, will return an erroneous result.

Note `OSMsgQEmpty()` performs pointer subtraction when computing the available room in the specified message queue. On some[90] targets, this may result in very slow execution. Since interrupts are disabled during `OSMsgQEmpty()`, this is not desirable. `OSMsgQCount()` always executes very quickly, and is preferred in these cases.

In the first example below, mainline code signals a message queue with a message from the user's `msg` array only if space is available. If not, an error counter is incremented. This example will give erroneous results if messages are also signaled to the same message queue from within an interrupt handler. That's because interrupts

[90] For example, on an 8-bit target where data pointers are 16 bits.

are enabled between the call to `OSMsgQEmpty()` and the call to `OSSignalMsgQ()`. In that case, `OSSignalMsgQ()`'s return code of `OSERR_EVENT_FULL` can be used to detect the inability to enqueue a message into a message queue.

In the second example below, the message queue is filled to capacity with new message pointers of ascending value, starting at 0.

See Also

`OS_WaitMsgQ()`, `OSCreateMsgQ()`, `OSMsgQCount()`, `OSReadMsgQ()`, `OSSignalMsgQ()`, `OSTryMsgQ()`

Example #1

```
#define MSGQ3_P OSECBP(4)

unsigned int counter;

if ( OSMsgQEmpty(MSGQ3_P) )
  OSSignalMsgQ(MSGQ3_P, (OStypeMsgP) &msg[i]);
else
  counter++;
```

Example #2

```
OStypeMsgQSize roomLeft;

roomLeft = OSMsgQEmpty(MSGQ1_P);
for ( i = 0 ; i < roomLeft; i++ )
      OSSignalMsgQ(MSGQ1_P, (OStypeMsgP) i);
```

OSReadBinSem(): Obtain a Binary Semaphore Unconditionally

Type:	Function
Prototype:	`OStypeBinSem OSReadBinSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`binsem.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the binary semaphore specified by `ecbP`.
Parameters:	`ecbP`: a pointer to the binary semaphore's ecb.
Returns:	Binary semaphore (0 or 1).
Stack Usage:	1

Notes

`OSReadBinSem()` has no effect on the specified binary semaphore. Therefore it can be used to obtain the binary semaphore's value without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadBinSem()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

In the example below, a binary semaphore employed as a resource is tested before making a decision to delay a task.

See Also

`OS_WaitBinSem()`, `OSCreateBinSem()`, `OSTryBinSem()`, `OSSignalBinSem()`

Example

```
…
/* initially, resource #2 is available.        */
OSCreateBinSem(BINSEM_RSRC2_P, 1);

void TaskD ( void )
{
  for (;;) {
    …
    if ( OSReadBinSem(BINSEM_RSRC2_P) )
      MyFn();
    else
      OS_Delay(100, TaskD1);
  }
}
```

OSReadEFlag(): Obtain an Event Flag Unconditionally

Type:	Function
Prototype:	`OStypeEFlag OSReadEFlag ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`eflag.c`
Enabled by:	`OSENABLE_EVENT_FLAGS`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the event flag specified by `ecbP`.
Parameters:	`ecbP`: a pointer to the event flag's ecb.
Returns:	Event flag.
Stack Usage:	1

Notes

`OSReadEFlag()` has no effect on the specified event flag. Therefore it can be used to obtain the event flag's value without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadEFlag()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than an event flag, will return an erroneous result.

In the example below, `TaskF()` waits on one of two bits to be set in an event flag pointed to by `EFLAG_P`. `OSReadEFlag()` is then used to determine which of the two bits was set.

See Also

`OS_WaitEFlag()`,`OSClrEFlag()`, `OSCreateEFlag()`, `OSSetEFlag()`

Example

```
void TaskF ( void )
{
  OStypeEFlag eFlag;


  for (;;) {
    OS_WaitEFlag(EFLAG_P, 0xC0, OSANY_BITS,
      OSNO_TIMEOUT, TaskF1);

    eFlag = OSReadEFlag(EFLAG_P);

    if ( eFlag & 0x80 )
      /* topmost bit was set …                */
      …
    else
      /* other bit was set …                  */
      …
  }
}
```

OSReadMsg():Obtain a Message's Message Pointer Unconditionally

Type:	Function
Prototype:	`OStypeMsgP OSReadMsg ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`msg.c`
Enabled by:	`OSENABLE_MESSAGES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns a pointer to the message specified in `ecbP`.
Parameters:	`ecbP`: a pointer to the message's ecb.
Returns:	Message pointer.
Stack Usage:	1

Notes

`OSReadMsg()` has no effect on the specified message. Therefore it can be used to obtain the message's message pointer without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadMsg()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a message, will return an erroneous result.

In the example below, a task checks to see if a message is non-empty before signaling the message.[91] Thus it avoids losing the message.

See Also

`OS_WaitMsg()`, `OSCreateMsg()`, `OSSignalMsg()`, `OSTryMsg()`

[91] If the application allowed signaling the message from an interrupt, additional interrupt control would be required in `TaskC()` in order to guarantee that the message is empty before signaling it.

Example

```
/* send this when there's a problem.          */
const char strImpMsg[] = "Important Message!\n";

void TaskC ( void )
{
  for (;;) {
    …
    /* delay one system tick as long as MSG    */
    /*  has a message in it.                   */
    while ( OSReadMsg(MSG_P) )
      OS_Delay(1, TaskC1);

    /* now that MSG is empty, we can send our  */
    /*  important message.                     */
    OSSignalMsg (MSG_P, (OStypeMsgP) &strImpMsg);
  }
}
```

OSReadMsgQ(): Obtain a Message Queue's Message Pointer Unconditionally

Type:	Function
Prototype:	`OStypeMsgP OSReadMsgQ ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`msgq.c`
Enabled by:	`OSENABLE_EVENT_READING`, `OSENABLE_MESSAGE_QUEUES`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns a pointer to the next message in the message queue specified in `ecbP`. Can also return the number of messages in the message queue.
Parameters:	`ecbP`: a pointer to the message's ecb.
Returns:	Message pointer.
Stack Usage:	1

Notes

`OSReadMsgQ()` has no effect on the specified message queue. Therefore it can be used to obtain the message queue's message pointer without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadMsgQ()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a message queue, will return an erroneous result.

In the example below, message queue #2 is slowly filled with a new character message every few seconds. `TaskB()` monitors the message queue every second. Whenever there are one or more valid messages in the message queue, `TaskB()` displays the first message's contents.[92] As the waiting task (not shown) waits the message queue and obtains the messages, `TaskB()`'s output will change as well.

See Also

`OS_WaitMsgQ()`, `OSCreateMsgQ()`, `OSSignalMsgQ()`, `OSTryMsgQ()`

Example

```
/* message queue #2 contains single chars.     */
```

[92] Note that `TaskB()`, as written, cannot distinguish between successive, identical messages. Therefore it will report on a stream of messages 'h','e','l','l','o' as 'h','e','l','o'. However, the waiting task will receive all five characters in the string.

```
#define MSGQ2_P OSECBP(6)

void TaskB ( void )
{
  static char oldchar;
  char newchar;
  OStypeMsgP msgP;

  for (;;) {
    OS_Delay(ONE_SEC, TaskB1);
    …

    /* test message queue #2                    */
    msgP = OSReadMsgQ(MSGQ2_P);

    /* get the message if there is one.         */
    if ( msgP ) {
      newchar = *(char *) msgP;
      if ( newchar != oldchar ) {
        oldchar = newchar;
        printf("The new message is: %c\n.",
          newchar);
      }
    }
    …
  }
}
```

OSReadSem(): Obtain a Semaphore Unconditionally

Type:	Function
Prototype:	`OStypeSem OSReadSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`sem.c`
Enabled by:	`OSENABLE_EVENT_READING`, `OSENABLE_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the current value of the semaphore specified in `ecbP`.
Parameters:	`ecbP`: a pointer to the semaphore's ecb.
Returns:	Semaphore.
Stack Usage:	1

Notes

`OSReadSem()` has no effect on the specified semaphore. Therefore it can be used to obtain the semaphore's value without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadSem()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a semaphore, will return an erroneous result.

In the example below, a binary semaphore is used to manage a 15-character ring buffer. In case of an error, the program displays a descriptive message[93] before re-initializing the buffer.

See Also

`OS_WaitSem()`, `OSCreateSem()`, `OSSignalSem()`, `OSTrySem()`

[93] `printf()` does not use the system's `Tx` facilities.

Example

```
/* initially, Tx buffer has room for 15 chars. */
#define SIZEOF_TXBUFF 15
…
/* manage the Tx buffer as a resource.         */
OSCreateSem(SEM_TXBUFF_P, SIZEOF_TXBUFF);


…
/* if there's a Tx error, flush and recreate   */
/*  the buffer after displaying a message.     */
if ( TxErr ) {
  DisableTxInts();
  printf("Error: %d chars stuck in Tx buffer.\n",
    SIZEOF_TXBUFF - OSReadSem(SEM_TXBUFF_P));
  FlushTxBuff();
  OSCreateSem(SEM_TXBUFF_P, SIZEOF_TXBUFF);
  EnableTxInts();
  }
…
```

OSRpt(): Display the Status of all Tasks, Events, Queues and Counters

Type:	Function
Prototype:	`void OSRpt ( void );`
Callable from:	Task or Background
Contained in:	`rpt.c`
Enabled by:	–
Affected by:	`OSBYTES_OF_COUNTS`, `OSBYTES_OF_DELAYS`, `OSENABLE_STACK_CHECKING`, `OSENABLE_STATISTICS`, `OSMON_HIDE_INVALID_PTRS`, `OSMON_SHOW_ONLY_ACTIVE`, `OSMON_SHOW_TOTAL_DELAY`, `OSUSE_EVENT_TYPES`
Description:	Display the current status of all Salvo tasks, events and counters in tabular form.
Parameters:	–
Returns:	–
Stack Usage:	3 + `printf()`'s stack usage

Notes

`OSRpt()` requires a working `printf()` function in the target application.[94] `OSRpt()` is quite large and is intended for use only in those systems that have sufficient code space (e.g. x86-based systems) to include it in the target application.

`OSRpt()` displays the current task, the members of the eligible and delayed queues (shown in their priority order), and the fields of each task control block (tcb) and event control block (ecb). If so configured, it also displays error, warning and timeout counter values, the maximum call ... return depth, and the total delay of the tasks in the delay queue.

`OSRpt()` reads and displays Salvo's data structures on-the-fly, i.e. no local copy is made. Depending on the speed at which the `printf()` function is able to output characters, `OSRpt()` may take quite a while to complete. This may result in a display of information that appears to be contradictory (e.g. a task is shown in the eligible queue and simultaneously waiting for an event). In order to

[94] Some libraries (e.g. Hi-Tech PICC) contain a dummy `putch()` function called by `printf()`. You must supply your own, working `putch()` for `printf()` output to occur.

avoid this, your application must control or disable interrupts while `OSRpt()` is executing.

See Also

Chapter 5 • Configuration

Example

```
…
/* display the current status of all tasks     */
/*  and events (and counters, if so enabled)   */
/*  to the system's terminal screen.           */
OSRpt();
…
```

A call to `OSRpt()` resulted in the following display on a simple terminal program connected via RS-232 to a Salvo system[95] with a working `printf()`:

```
Salvo v2.2.beta7  Max call...rtn stack depth: 3
CtxSws, total=idle+eligible: 1000358326 = 922445444 + 77912882
Errors: 0  Warnings: 0  Timeouts: 255  Ticks: 33163186
EligQ: t6,t3,t8
DelayQ: t7,t1,t2,t5,t4  Total delay: 60 ticks
task stat prio    addr   t->  e->  d-> delay
  1  wait   2      6F8h    .  e 1  t 2   22
  2  wait   3      6F8h    .  e 2  t 5   10
  3  elig   4      6FBh  t 8  n/a
  4  wait   5      6F8h    .  e 4    .   13
  5  wait   5      6F8h  t 4  e 4  t 4   13
  6  elig   1      6F5h  t 3  n/a
  7  dlyd   0      70Ah    .       t 1    2
  8  elig  15      70Dh    .  n/a

evnt type t->    value
  1   Sem t 1        0
  2   Sem t 2        0
  3   Sem   .        0
  4   Sem t 5        0
  5   Sem   .      255
```

Figure 51: OSRpt() Output to Terminal Screen

In Figure 51 we can see that when `OSRpt()` was called, three tasks were eligible, five were waiting and/or delayed, and over one billion context switches had occurred over a nearly four-day-long period.[96]

95 This output is from the program in `salvo\demo\d1\sysa`, running on a PIC16C77 with a 4MHz crystal.

96 System tick rate of 100Hz.

OSSched(): Run the Highest-Priority Eligible Task

Type:	Function
Prototype:	`void OSSched ( void );`
Callable from:	`main()`
Contained in:	`sched.c`
Enabled by:	–
Affected by:	`OSCLEAR_UNUSED_POINTERS`, `OSCLEAR_WATCHDOG_TIMER`, `OSENABLE_STACK_CHECKING`, `OSENABLE_STATISTICS`, `OSLOGGING`, `OSOPTIMIZE_FOR_SPEED`,
Description:	Dispatch Salvo's tasks via a cooperative multitasking priority-based scheme.
Parameters:	–
Returns:	–
Stack Usage:	2 if `OSUSE_INLINE_OSSCHED` is `FALSE`. Tasks will run 2 levels below scheduler. 1 if `OSUSE_INLINE_OSSCHED` is `TRUE`. Tasks will run 1 level below scheduler.

Notes

`OSSched()` causes the highest-priority task currently in the eligible queue to execute.

Your application must call `OSInit()` before calling `OSSched()`.

Your application must repeatedly call `OSSched()` in order for multitasking to continue.

In the example below, `OSSched()` is called from within an infinite loop.

See Also

`OSCreateTask()`, `OSInit()`, `OSStartTask()`

Example

```
main()
{
  /* OS must be initialized.                    */
  OSInit();
  …
  /* create and start several tasks ...         */
  OSCreateTask(Task0, OSTCBP(1), TASK0_PRIORITY);
  OSCreateTask(Task1, OSTCBP(2), TASK1_PRIORITY);
  …
  /* tasks are ready to run – begin multi-      */
  /*  tasking.                                  */
  for (;;) {
    /* OSSched() is usually the only function  */
    /*  called inside this never-ending loop.  */
    OSSched();
  }
}
```

OSSetEFlag(): Set Event Flag Bit(s)

Type:	Macro or Function
Prototype:	`OStypeErr OSSetEFlag ( OStypeEcbP ecbP, OStypeEFlag mask );`
Callable from:	Anywhere
Contained in:	`eflag.c`, `event.c`
Enabled by:	`OSENABLE_EVENT_FLAGS`, `OSEVENTS`
Affected by:	`OSLOGGING`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSUSE_EVENT_TYPES`
Description:	Set bits in an event flag. If any bits change, every task waiting it is made eligible.
Parameters:	`ecbP`: a pointer to the event flag's ecb. `mask`: mask of bits to be set.
Returns:	`OSERR_BAD_P` if event flag pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not an event flag. `OSERR_EVENT_CB_UNINIT` if event flag's control block is uninitialized. `OSERR_EVENT_FULL` if event flag doesn't change. `OSNOERR` if event flag bits are successfully set.
Stack Usage:	1

Notes

All tasks[97] waiting an event flag are made eligible by forcing any zeroed bits to one in the event flag via `OSSetEFlag()`. Upon running, each such task will either continue running or will return to the waiting state, depending on the outcome of its call to `OS_WaitEFlag()`. Thus, multiple tasks waiting a single event flag can be made eligible simultaneously.

In the example below, two tasks are each waiting different bits of an event flag. When those bits are set via `OSSetEFlag()`, both tasks are made eligible. Each task will run when it becomes the highest-priority eligible task.

[97] Not just the highest-priority waiting task.

See Also

OS_WaitEFlag(), OSClrEFlag(), OSCreateEFlag(), OSReadEFlag()

Example

```
#define EFLAG2_P OSECBP(4)
…
/* force TaskA() and TaskB() to wake up.       */
OSSetEFlag(EFLAG2_P, 0x03);
…
void TaskA( void )
{
  for (;;) {
    /* wait forever for bit 0 to be set        */
    OS_WaitEFlag(EFLAG2_P, 0x01, OSALL_BITS,
      OSNO_TIMEOUT, TaskA1);

    /* clear it and continue                   */
    OSClrEFlag(EFLAG2_P, 0x01);

    …
  }
}

void TaskB( void )
{
  for (;;) {
    OS_WaitEFlag(EFLAG2_P, 0x02, OSALL_BITS,
      OSNO_TIMEOUT, TaskA1);
    OSClrEFlag(EFLAG2_P, 0x02);
    …
  }
}
```

OSSetPrio(): Change the Current Task's Priority

Type:	Function
Prototype:	`void OSSetPrio ( OStypePrio prio );`
Callable from:	Task only
Contained in:	`prio.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Change the priority of the current (running) task.
Parameters:	`priority`: the desired (new) priority for the current task.
Returns:	–
Stack Usage:	1

Notes

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest.

Tasks can share priorities. Eligible tasks with the same priority will round-robin schedule as long as they are the highest-priority eligible tasks.

The new priority will take effect immediately after the next context switch.

In the example below, `TaskStatusLED()` is dedicated to flashing an LED at one of two rates – 1Hz for a simple heartbeat indication, and 25Hz for an alert indication. The system timer ticks every 10ms. When an alert is not present, it's sensible to run `TaskStatusLED()` at a low priority, so that other more important tasks can run. However, when an alert condition occurs, it's imperative that the user see the LED flash at 25Hz, so `TaskStatusLED()` elevates itself to a higher priority to ensure that it runs often enough to flash the LED at 25Hz. This example assumes that all other tasks are either delayed or waiting at any particular time. Note that in this example `TaskStatusLED()` will fail to flash the LED at 25Hz if it is blocked (i.e. if there are always higher-priority tasks running) at priority 14 when alert is `TRUE`.

See Also

`OS_SetPrio()`, `OSGetPrio()`, `OSGetPrioTask()`, `OSSetPrioTask()`, `OSDISABLE_TASK_PRIORITIES`

Example

```
char alert = FALSE;  /* global, set & reset    */
                     /* elsewhere in code      */

void TaskStatusLED( void )
{
  for (;;) {
    /* toggle alert LED                        */
    PORT_LED ^= 0x01;

    /* if there's an alert, elevate the task's */
    /*  priority (to ensure that we see the LED*/
    /*  flash) and change the flash rate to    */
    /*  25Hz to be sure to catch the user's    */
    /*  attention.                             */
    if ( alert ) {
      OSSetPrio(5);
      OS_Delay(2, TaskStatusLED1);
    }

    /* otherwise  lower the task's priority to */
    /*  rock-bottom and toggle the LED at 1Hz. */
    else {
      OSSetPrio(OSLOWEST_PRIO);
      OS_Delay(50, TaskStatusLED2);
    }
  }
}
```

OSSetPrioTask(): Change a Task's Priority

Type:	Function
Prototype:	`OStypeErr OSSetPrioTask ( OStypeTcbP tcbP, OStypePrio prio );`
Callable from:	Task or Background
Contained in:	`task.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Change the priority of the specified task.
Parameters:	`tcbP`: a pointer to the task's tcb. `prio`: the desired (new) priority for the specified task.
Returns:	`OSNOERR` if specified task's priority was changed successfully `OSERR` if `OSSetPrioTask()` was unable to change the specified task's priority.
Stack Usage:	3

Notes

`OSSetPrioTask()` can change the priority of tasks that are stopped (`OSTA_STOPPED`), delayed (`OSTA_DELAYED`), eligible (`OSTA_ELIGIBLE`) or have timed out (`OSTA_TIMEDOUT`). It cannot change the priority of tasks in other states.

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest.

Tasks can share priorities. Eligible tasks with the same priority will round-robin schedule as long as they are the highest-priority eligible tasks.

The new priority will take effect immediately.

In the example below, every ten minutes `TaskE()` elevates the priority of `TaskC()`for one minute, then reduces `TaskC()`'s priority back to its original priority.

See Also

`OSGetPrioTask()`, `OSDISABLE_TASK_PRIORITIES`

Example

```
/* initially, run TaskD() at priority 7.       */
OSCreateTask(TaskD, TASKD_P, 7);
OSCreateTask(TaskE, TASKE_P, 3);

void TaskE( void )
{
  for (;;) {
    /* delay ten minutes.                       */
    OS_Delay(TEN_MINUTES, TaskE1);

    /* elevate TaskD()'s priority.              */
    OSSetPrioTask(TASKD_P, 5);

    /* delay another minute.                    */
    OS_Delay(ONE_MINUTE, TaskE2);

    /* restore TaskD()'s priority.              */
    OSSetPrioTask(TASKD_P, 7);
  }
}
```

OSSetTicks(): Initialize the System Timer

Type:	Function
Prototype:	`void OSSetTicks ( OStypeTick tick );`
Callable from:	Anywhere
Contained in:	`ticks.c`
Enabled by:	`OSBYTES_OF_TICKS`
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	(Re-)define the current value of the system timer (in ticks).
Parameters:	`tick`: an integer (>=0) value for the system timer.
Returns:	–
Stack Usage:	1

Notes

The system timer is initialized to 0 via `OSInit()`.

In the example below, the current value of the system timer is reset to zero during runtime.

See Also

`OSGetTicks()`

Example

```
…

/* reset system ticks to 0.                        */
OSSetTicks(0);

…
```

On certain targets it may be advantageous to write the current system ticks (`OStimerTicks`) directly instead of through `OSSetTicks()`. Possible scenarios include substantial function call overhead and/or no need to manage interrupts. In the example below, the current value of the system timer is reset to zero during runtime.

```
…

/* reset system ticks to 0.                        */
OSDi();
OStimerTicks = 0;
OSEi();

…
```

OSSetTS(): Initialize the Current Task's Timestamp

Type:	Macro (invokes `OSSetTSTask()`)
Prototype:	`void OSSetTS ( OStypeTS timestamp );`
Callable from:	Task only
Contained in:	`delay3.c`
Enabled by:	`OSBYTES_OF_TICKS`
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	(Re-)define the current task's timestamp (in ticks).
Parameters:	`timestamp`: an integer (>=0) value for the timestamp.
Returns:	–
Stack Usage:	1

Notes

When a task is created, its timestamp is initialized to an `OStypeTS`-sized version of the system timer ticks, i.e. `(OStypeTS) OStimerTicks`.

In the example below, the task resets its timestamp upon starting. It then preserves its timestamp prior to invoking `OS_Delay()` as part of a hardware initialization sequence. Thereafter, it will time out every 6 ticks relative to when it started. If `OS_Delay()` had been used, it would time out every six ticks relative to when `OS_Delay()` was called.

See `OS_DelayTS()` for more information on timestamps.

See Also

`OS_DelayTS()`, `OSGetTS()`, `OSSyncTS()`

Example

```
void Task( void )
{
  OStypeTS timestamp;


  /* synchronize delays with the start of this */
  /*  task, i.e. timestamp = now.              */
  OSSetTS((OStypeTS) OSGetTicks());

  /* do various things here.                   */
  …
  OS_Yield(label1);
  …

  /* initialize some peripheral that requires  */
  /*  a short delay. Must preserve timestamp   */
  /*  when calling OS_Delay().                 */
  …
  timestamp = OSGetTS();
  OS_Delay(1, label2);
  OSSetTS(timestamp);
  /* continue initializing said peripheral.    */
  …

  while ( TRUE ) {
    /* as long as no more than 5 ticks have    */[98]
    /*  passed since this task was started,    */
    /*  the task will timeout at timestamp + 6 */
    /*  ticks, and then timestamp + 12, + 18,  */
    /*  etc.                                   */
    OS_DelayTS(6, label3);
    …
  }
}
```

[98] 5 ticks because of the system timer's inherent +/- 1 tick accuracy.

OSSignalBinSem(): Signal a Binary Semaphore

Type:	Macro or Function
Prototype:	`OStypeErr OSSignalBinSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`binsem.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Signal a binary semaphore. If one or more tasks are waiting for the semaphore, the highest-priority task is made eligible.
Parameters:	`ecbP`: a pointer to the semaphore's ecb.
Returns:	`OSERR_BAD_P` if binary semaphore pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not a binary semaphore. `OSERR_EVENT_FULL` if binary semaphore is already 1. `OSNOERR` on success.
Stack Usage:	1

Notes

No more than one task can be made eligible by signaling a binary semaphore.

In the example below, a binary semaphore is used to signal a waiting task. `TaskWaveformGenerator()` outputs an 8-bit waveform to a DAC whenever it receives a signal to do so. The binary semaphore is initialized to 0, so `TaskWaveformGenerator()` remains in the waiting state until the `BINSEM_GEN_WAVEFORM` is signaled elsewhere in the program, whereupon it outputs an array of 8-bit values to a port. It then resumes waiting until `BINSEM_GEN_WAVEFORM` is signaled again.

See Also

`OS_WaitBinSem()`, `OSCreateBinSem()`, `OSReadBinSem()`, `OSTryBinSem()`

Example

```
…

#define BINSEM_GEN_WAVEFORM_P OSECBP(5)

…

OSCreateBinSem(BINSEM_GEN_WAVEFORM_P, 0);

…

/* tell waveform-generating task to create a   */
/*  single waveform.                           */
OSSignalBinSem(BINSEM_GEN_WAVEFORM_P);

…

void TaskWaveformGenerator( void )
{
  char i;

  for (;;) {
    /* wait forever for signal to generate     */
    /*  waveform.                              */
    OS_WaitBinSem(BINSEM_GEN_WAVEFORM_P,
      OSNO_TIMEOUT, TaskWaveformGenerator1);

    /* output waveform to DAC.                 */
    for ( i = 0 ; i < 256 ; i ++ )
      DACPORT = WAVEFORM_TABLE[i];
  }
}
```

OSSignalMsg(): Send a Message

Type:	Macro or Function
Prototype:	`OStypeErr OSSignalMsg ( OStypeEcbP ecbP, OStypeMsgP msgP );`
Callable from:	Anywhere
Contained in:	`msg.c`
Enabled by:	`OSENABLE_MESSAGES`, `OSEVENTS`
Affected by:	`OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Signal a message with the value specified. If one or more tasks are waiting for the message, the highest-priority task is made eligible.
Parameters:	`ecbP`: a pointer to the message's ecb. `msgP`: a pointer to a message.
Returns:	`OSERR_BAD_P` if message pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not a message. `OSERR_EVENT_FULL` if message is already defined. `OSNOERR` on success.
Stack Usage:	1

Notes

No more than one task can be made eligible by signaling a message.

In the example below, a message is used (in place of a binary semaphore) to control access to a shared resource, an LCD. When either `TaskDisplay()` or `TaskFlashWarning()` needs to write to the display, it must first acquire the display by successfully waiting on the message `MSG_LCD_RSRC`. Once obtained, the task can write to the LCD. When finished, it must release the resource by signaling the message.

`TaskFlashWarning()` displays a warning message for five seconds by writing to the display and then delaying itself for five seconds before releasing the resource. The use of a message to control access to the LCD prevents `TaskDisplay()` from overwriting the LCD while the warning message is displayed.

See Also

OS_WaitMsg(), OSCreateMsg(), OSReadMsg(), OSTryMsg()

Example

```
#define MSG_DISP_UPDATE_P OSECBP(2)    /* flag */
#define MSG_LCD_RSRC_P    OSECBP(3)    /* rsrc */
#define MSG_WARNING_P     OSECBP(4)    /* flag */

char strLCD[LCD_LENGTH+1]; /* 1 row chars + \0 */

void TaskDisplay( void )
{
  static OStypeMsgP msgP;

  /* display is initially available to all.    */
  OSCreateMsg(MSG_LCD_RSRC_P, (OStypeMsgP) 1);

  for (;;) {
    /* wait until display update is required   */
    OS_WaitMsg(MSG_DISP_UPDATE_P, &msgP,
      OSNO_TIMEOUT, TaskDisplay1);

    /* wait if we can't acquire the resource.  */
    OS_WaitMsg(MSG_LCD_RSRC_P, &msgP,
      OSNO_TIMEOUT, TaskDisplay2);

    /* write global string to display.         */
    WriteLCD(strLCD);

    /* free display for others to use.         */
    OSSignalMsg(MSG_LCD_RSRC_P, (OStypeMsgP) 1);
  }
}

void TaskFlashWarning( void )
{
  static OStypeMsgP msgP, msgP2;

  for (;;) {
    /* wait for the warning ...                */
    OS_WaitMsg(MSG_WARNING_P, &msgP,
      OSNO_TIMEOUT, TaskFlashWarning1);

    /* grab the LCD, locking others out.       */
    OS_WaitMsg(MSG_LCD_RSRC_P, &msgP2,
      OSNO_TIMEOUT, TaskFlashWarning2);

    /* Flash warning on LCD for 5 seconds.     */
    WriteLCD((char *)msgP);
    OS_Delay(FIVE_SEC, TaskFlashWarning3);

    /* refresh / restore LCD, and free it.     */
    WriteLCD(strLCD);
    OSSignalMsg(MSG_LCD_RSRC_P, (OStypeMsgP) 1);
  }
}
```

OSSignalMsgQ(): Send a Message via a Message Queue

Type:	Macro or Function
Prototype:	`OStypeErr OSSignalMsgQ ( OStypeEcbP ecbP, OStypeMsgP msgP );`
Callable from:	Anywhere
Contained in:	`msgq.c`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`, `OSEVENTS`
Affected by:	`OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Send a message to a task via the message queue specified with ecbP. If one or more tasks are waiting the message queue, the highest-priority task is made eligible.
Parameters:	`ecbP`: a pointer to the message queue's ecb. `msgP`: a pointer to a message.
Returns:	`OSERR_BAD_P` if message queue pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not a message queue. `OSERR_EVENT_CB_UNINIT` if the message queue's control block is uninitialized. `OSERR_EVENT_FULL` if message queue is full. `OSNOERR` on success.
Stack Usage:	1

Notes

No more than one task can be made eligible by signaling a message.

In the example below, `Commands[]` is a constant array of one-character commands. A message queue is used to send multiple commands to a waiting task. The two successive calls to `OSSignalMsg()` will place the HALT ('h') and EXIT ('x') commands into the message queue, but only if room is available. Upon arrival of the messages, the receiving task will act accordingly.

See Also

`OS_WaitMsgQ()`, `OSCreateMsgQ()`, `OSReadMsgQ()`, `OSTryMsgQ()`

Example

```
const char Commands[4] = { 'a', 'g', 'h', 'x' };
…

OSSignalMsgQ(MSGQ5_P, (OStypeMsgP) &Commands[2]);
OSSignalMsgQ(MSGQ5_P, (OStypeMsgP) &Commands[3]);
…
```

OSSignalSem(): Signal a Semaphore

Type:	Macro or Function
Prototype:	`OStypeErr OSSignalSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`sem.c`
Enabled by:	`OSENABLE_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSBIG_SEMAPHORES`, `OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Increment a counting semaphore. If one or more tasks are waiting for the semaphore, the highest-priority task is made eligible.
Parameters:	`ecbP`: a pointer to the semaphore's ecb.
Returns:	`OSERR_BAD_P` if semaphore pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not a semaphore. `OSERR_EVENT_FULL` if semaphore is already at its maximum allowed value. `OSNOERR` on success.
Stack Usage:	1

Notes

No more than one task can be made eligible by signaling a semaphore.

8- or 16-bit semaphores can be selected via the `OSBIG_SEMAPHORES` configuration option.

In the example below, a counting semaphore is used to keep track of how many characters are waiting in the receive buffer `rxBuff`. Another task that waits on `SEM_RX_BUFF` will remove and process them, one at a time, from the buffer. By communicating between the tasks with a semaphore, the tasks can run at different priorities – `TaskRx()` can run at a high priority to ensure that the UART's receive buffer is not overrun, and the processing task (which waits on `SEM_RX_BUFF`) can run at a lower priority while parsing incoming command strings.

See Also

`OS_WaitSem()`, `OSCreateSem()`, `OSReadSem()`, `OSTrySem()`

Example

```
void TaskRx( void )
{
  /* initially there are no Rx chars for       */
  /* TaskRcvRsp() to process.                  */
  OSCreateSem(SEM_RX_RBUFF_P, 0);

  /* The task to interpret responses is driven */
  /*  solely by TaskRx()'s collecting incoming */
  /*  incoming chars for it, so we'll launch   */
  /*  it from here.                            */
  OSCreateTask(TaskRcvRsp, TASK_RCV_RSP_P,
    TASK_RCV_RSP_PRIO);

  /* deal with Rx chars. */
  for (;;) {
    /* if there are any Rx chars waiting,      */
    /*  signal the command interpreter.        */
     while ( SioRxQue(Port) > 0 ) {
      /* put new Rx char into local buffer     */
      rxBuff[rxTail] = (char) SioGetc(Port, 10);

      /* massage buffer pointers               */
      rxTail++;
      rxCount++;
      if ( rxTail >= SIZEOF_RX_BUFF )
        rxTail = 0;

      /* signal the command interpreter that   */
      /*  there's work to be done. In this     */
      /*  implementation we signal once for    */
      /*  every new character received.        */
      OSSignalSem(SEM_RX_RBUFF_P);

    } /* while ( SioRxQue(Port) > 0 )          */

  /* wait a while and poll again.              */
  OS_Delay(1, TaskRx1);
  }
}
```

OSStartTask(): Make a Task Eligible To Run

Type:	Function
Prototype:	`OStypeErr OSStartTask ( OStypeTcbP tcbP );`
Callable from:	Anywhere
Contained in:	`task.c`
Enabled by:	–
Affected by:	`OSLOGGING`, `OSENABLE_STACK_CHECKING`
Description:	Start the specified task.
Parameters:	`tcbP`: a pointer to the task's tcb.
Returns:	`OSNOERR` if task is successfully started. `OSERR` if either the specified tcb pointer is invalid (i.e. out-of-range), or if the specified task's state is not `OSTA_STOPPED`.
Stack Usage:	3

Notes

`OSStartTask()` can only start a task that is in the stopped (`OSTA_STOPPED`) state.

Starting a task simply places it into the eligible queue. It will not run until it becomes the highest-priority eligible task.

A task that has been started is in the eligible state.

A task must be created via `OSCreateTask()` before it can be started via `OSStartTask()`.

In the example below, `TaskToggleLED` is created but is only made eligible to run via the call to `OSStartTask()`. Without the call to `OSStartTask()`, the task would remain stopped indefinitely.

See Also

`OSCreateTask()`, `OSInit()`

Example

```
…

/* this task toggles an LED each time it       */
/*  runs, i.e. whenever it's the highest-      */
/*  priority eligible task.                    */
void TaskToggleLED( void )
{
  for (;;) {
    /* toggle LED on pin 0 of PORT B */
    PORTB ^= 0x01;

    OS_Yield(TaskToggleLED1);
  }
}

main()
{
  …

  /* create and start TaskToggleLED0() with    */
  /*  the lowest priority. We'll observe the   */
  /*  LED toggling when no other tasks are     */
  /*  eligible to run.                         */
  OSCreateTask(TaskToggleLED, OSTCBP(5),
    OSDONT_START_TASK | OSLOWEST_PRIO);

  …

  OSStartTask(OSTCBP(5));

  …

  for (;;)
    OSSched();
}
```

OSStopTask(): Stop a Task

Type:	Function
Prototype:	`OStypeErr OSStopTask ( OStypeTcbP tcbP );`
Callable from:	Task or Background
Contained in:	`task2.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Stop the specified task.
Parameters:	`tcbP`: a pointer to the task's tcb.
Returns:	`OSNOERR` if specified task was successfully stopped. `OSERR` if `OSStopTask()` was unable to stop the specified task.
Stack Usage:	3

Notes

`OSStopTask()` can stop tasks that are delayed (`OSTA_DELAYED`), eligible (`OSTA_ELIGIBLE`) or have timed out (`OSTA_TIMEDOUT`). It cannot stop tasks in other states. Tasks that are already stopped (`OSTA_STOPPED`) will remain stopped.

A stopped task can be restarted with `OSStartTask()`.

In the example below, `TaskStopBeep()` exists only to stop another task, `TaskBeep()`. `TaskStopBeep()` waits forever for the binary semaphore `BINSEM_STOP_BEEP` to be signaled. When this occurs, it calls `OSStopTask()`, which stops `TaskBeep()`. `TaskStopBeep()` then begins waiting the binary semaphore again. By setting `TaskStopBeep()`'s priority to be higher than `TaskBeep()`'s, `TaskStopBeep()` is able to stop `TaskBeep()` at the earliest opportunity.

This example also illustrates how program control can pass from an interrupt through a task and affect another task, even if `OSStopTask()` is not called from an interrupt. By calling `OSSignalBinSem(BINSEM_STOP_BEEP)` from an ISR, `TaskBeep()` will be stopped by `TaskStopBeep()` before its earliest opportunity to run again.

See Also

`OSStartTask()`, `OS_Stop()`

Example

```
OSCreateTask(TaskBeep,     TASK_BEEP_P,     7);
OSCreateTask(TaskStopBeep, TASK_STOPBEEP_P, 6);
OSCreateSem(BINSEM_STOP_BEEP_P, 0);
…
void TaskStopBeep ( void )
{
  for (;;) {
    OS_WaitBinSem(BINSEM_STOP_BEEP_P,
      OSNO_TIMEOUT, TaskStopBeep1);
    OSStopTask(TASK_BEEP_P);
  }
}
```

OSSyncTS(): Synchronize the Current Task's Timestamp

Type:	Macro (invokes `OSSyncTSTask()`)
Prototype:	`void OSSyncTS ( OStypeInterval interval );`
Callable from:	Task only
Contained in:	`delay2.c`
Enabled by:	–
Affected by:	`OSENABLE_DELAYS`, `OSENABLE_TICKS`
Description:	Synchronize the current task's timestamp against the current timer ticks.
Parameters:	`interval`: a signed offset relative to the current timer ticks.
Returns:	–
Stack Usage:	2

Notes

`OSSyncTS()` is used in conjunction with `OS_DelayTS()` to synchronize the current task's delays against an absolute value of the system's timer ticks. With `OSSyncTS()`, you can increment or decrement the value of current task's timestamp.[99]

In the example below, `TaskPeriodic()` begins by running every 16 system ticks. If the global variable `shiftTicks` is found to be non-zero, it is copied to a local variable `offset`, cleared, and then used to phase-shift `TaskPeriodic()` with a resolution of 1 system tick.

See Also

`OS_DelayTS()`, `OSGetTS()`, `OSSetTS()`

[99] Use `OSSetTS()` to change the absolute value of the current task's timestamp.

Example

```
OStypeInterval  shiftTicks;   /* -15 to +15    */
…
void TaskPeriodic ( void )
{
  OStypeInterval offset;

  for (;;) {
    OS_DelayTS(16, TaskPeriodic1);
    …
    if ( shift ) {
      OSDi();
      offset = shiftTicks;
      shiftTicks = 0;
      OSEi();
      OSSyncTS(offset);
    }
  }
}
```

OSTimer(): Run the Timer

Type:	Function
Prototype:	`void OSTimer ( void );`
Callable from:	Foreground (preferred) or background.
Contained in:	`timer.c`
Enabled by:	`OSBYTES_OF_DELAYS`, `OSBYTES_OF_TICKS`
Affected by:	`OSDISABLE_ERROR_CHECKING`, `OSENABLE_DELAYS`, `OSENABLE_STACK_CHECKING`, `OSENABLE_TICKS`, `OSTIMER_PRESCALAR`
Description:	Perform Salvo's timer-based services.
Parameters:	–
Returns:	–
Stack Usage:	2 if `OSUSE_INLINE_OSTIMER` is `FALSE`. 1 if `OSUSE_INLINE_OSTIMER` is `TRUE`.

Notes

If delay, elapsed time and/or timeout services are desired, `OSTimer()` must be called at the desired system tick rate. Context switching and event services do not require `OSTimer()` to be installed.

The rate at which `OSTimer()` is called by your application (typically every 5-100ms) must allow sufficient time for `OSTimer()` to complete its actions.

In the example below, the timer is called from within an interrupt service routine (ISR) as a periodic event. Each time `OSTimer()` is called it checks to see if any delayed or waiting tasks have timed out, and if so, re-enters them into the eligible queue.

`OSTimer()` is very small and is easily incorporated into an ISR without major deleterious effects.

Example

```
void interrupt ISR( void )
{
  /* OSTimer() is called on every timer0       */
  /*  interrupt.                               */

  if ( TOIF ) {
    /* must clear timer0 interrupt flag.       */
    TOIF = 0;

    /* let Salvo handle delays, ticks          */
    /*  and timeouts.                          */
    OSTimer();
  }

  /* handle other interrupt sources.           */
  …
}
```

OSTryBinSem(): Obtain a Binary Semaphore if Available

Type:	Function
Prototype:	`OStypeBinSem OSTryBinSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`binsem2.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the binary semaphore specified by `ecbP`. If the semaphore is 1, reset it to 0.
Parameters:	`ecbP`: a pointer to the binary semaphore's ecb.
Returns:	Binary semaphore (0 or 1).
Stack Usage:	1

Notes

`OSTryBinSem()` is like `OS_WaitBinSem()`, but it does not context-switch the current task if the binary semaphore is not available (i.e. has a value of 0). Therefore `OSTryBinSem()` can be used outside of the current task to obtain the binary semaphore, e.g. in an ISR.

No error checking is performed on the `ecbP` parameter. Calling `OSTryBinSem()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

In the example below, `TaskC()`has a higher priority than `TaskD()` and obtains the binary semaphore whenever it is set to 1. Signaling the binary semaphore does not change the state of `TaskC()`. As long as `TaskC()` is running, `TaskD()` will wait forever for the binary semaphore.[100]

See Also

`OS_WaitBinSem()`, `OSCreateBinSem()`, `OSReadBinSem()`, `OSSignalBinSem()`

[100] This assumes that `TaskD()` unsuccessfully waited the binary semaphore before `TaskC()` started running.

Example

```
/* priority of 3                                    */
void TaskC ( void )
{
  for (;;) {
    if ( OSTryBinSem(BINSEM2_P) )
      printf("binSem #2 was 1, now 0.\n");
    else
      printf("binSem #2 is 0.\n");

    OS_Yield(TaskC1);
    …
  }
}

/* priority of 9 (lower)                            */
void TaskD ( void )
{
  for (;;) {
    OS_WaitBinSem(BINSEM2_P,
      OSNO_TIMEOUT, TaskD1);
    …
  }
}
```

OSTryMsg(): Obtain a Message if Available

Type:	Function
Prototype:	`OStypeMsg OSTryMsg ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`msg2.c`
Enabled by:	`OSENABLE_MESSAGES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns a pointer to the message specified by `ecbP`. If the message exists, the message's own pointer is cleared.
Parameters:	`ecbP`: a pointer to the message's ecb.
Returns:	Message pointer.
Stack Usage:	1

Notes

`OSTryMsg()` is like `OS_WaitMsg()`, but it does not context-switch the current task if the message is not available (i.e. the message pointer has a value of 0). Therefore `OSTryMsg()` can be used outside of the current task to obtain the message, e.g. in an ISR.

No error checking is performed on the `ecbP` parameter. Calling `OSTryMsg()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

Waiting on a message (i.e. via `OS_WaitMsg()`) is not permitted within an interrupt service routine. In the example below, `OSTryMsg()` is used within the ISR in order to obtain a message without waiting. Regardless of whether or not a message was available, the message will be empty at the end of the ISR.

See Also

`OS_WaitMsg()`, `OSCreateMsg()`, `OSReadMsg()`, `OSSignalMsg()`

Example

```
void interrupt myISR ( void )
{
  OStypeMsgP msgP;

  /* get message pointer (may be 0).          */
  msgP = OSTryMsg(MSG3_P);

  if ( msgP ) {
    /* do something with the message.         */
    …
  }
  else {
    /* message wasn't available.              */
    …
  }
  …
}
```

OSTryMsgQ(): Obtain a Message from a Message Queue if Available

Type:	Function
Prototype:	`OStypeMsgQ OSTryMsgQ ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`msgq2.c`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns a pointer to the first available message in the message queue specified by `ecbP`. If the message queue contains any messages, remove the message from the queue.
Parameters:	`ecbP`: a pointer to the message queue's ecb.
Returns:	Message pointer.
Stack Usage:	1

Notes

`OSTryMsgQ()` is like `OS_WaitMsgQ()`, but it does not context-switch the current task if the message queue is empty. Therefore `OSTryMsgQ()` can be used outside of the current task to obtain the message in the message queue, e.g. in an ISR.

No error checking is performed on the `ecbP` parameter. Calling `OSTryMsgQ()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

In the example below, after each call to the scheduler, a `char` message is removed from a message queue and then re-inserted. As long as no services involving this message queue are called from within an interrupt, this will rotate the order of the messages in the message queue indefinitely. For example, a message queue containing the four single-character messages 's', 't', 'o' and 'p' becomes 't', 'o', 'p' and 's'.

See Also

`OS_WaitMsgQ()`, `OSCreateMsgQ()`, `OSReadMsgQ()`, `OSSignalMsgQ()`

Example

```
OStypeMsgP msgP;
…
for (;;) {
  OSSched();

  msgP = OSTryMsgQ(MSGQ3_P);

  if ( msgP ) {
    printf("removed message %c from msgQ.\n",
      *(char *) msgP);

    OSSignalMsgQ(MSGQ3_P, msgP);

    printf("re-inserted message into msgQ.\n");
  }
}
```

OSTrySem(): Obtain a Semaphore if Available

Type:	Function
Prototype:	`OStypeSem OSTrySem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`sem2.c`
Enabled by:	`OSENABLE_SEMAPHORES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the semaphore specified by `ecbP`. If the semaphore is non-zero, decrement it.
Parameters:	`ecbP`: a pointer to the semaphore's ecb.
Returns:	Semaphore.
Stack Usage:	1

Notes

`OSTrySem()` is like `OS_WaitSem()`, but it does not context-switch the current task if the semaphore is not available (i.e. has a value of 0). Therefore `OSTrySem()` can be used outside of the current task to obtain the semaphore, e.g. in an ISR.

No error checking is performed on the `ecbP` parameter. Calling `OSTrySem()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

In the example below, `OSTrySem()` is used by `FlushBuffer()`[101] to flush a buffer that is managed through a counting semaphore. Afterwards, `i` holds the count of the items that were in the the buffer before it was flushed.

See Also

`OS_WaitSem()`, `OSCreateSem()`, `OSReadSem()`, `OSSignalSem()`

[101] Note that `FlushBuffer()` is a simple function, and not a task. The flushing operation could also be performed in a task.

Example

```
/* buffer is initially empty.                    */
OSCreateSem(SEM2_P, 0);
…
void FlushBuffer ( void )
{
  char i;

  /* count and remove the buffer's contents.    */
  i = 0;
  while ( OSTrySem(SEM2_P) )
    i++;
}
```

Additional User Services

OSAnyEligibleTasks (): Check for Eligible Tasks

Type:	Macro
Declaration:	`OSAnyEligibleTasks()`
Callable from:	Outside `OSSched()` (background) or inside a task or its subroutines.
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	–
Description:	Detect if any tasks are currently eligible to run.
Parameters:	–
Returns:	`TRUE` if one or more tasks are eligible, `FALSE` otherwise.
Stack Usage:	0

Notes

`OSAnyEligibleTasks()` cannot predict when waiting and/or delayed tasks will become eligible. This must be considered when using `OSAnyEligibleTasks().`

`OSAnyEligibleTasks()` returns `FALSE` if a task is running and no tasks are eligible.

In the first example below, a Salvo application's main loop has been modified to run an alternative process (e.g. some legacy code written in assembler) in addition to the scheduler. This alternative process must terminate within a short time in order to avoid problems scheduling tasks. By invoking the alternative process only when no tasks are eligible, it can "steal cycles" that the scheduler does not currently need.

In the second example, a user *function* (not a task) is called only when the system is idling, i.e. when tasks are eligible to run. This idling function must execute quickly so as not to affect task execution.

Example #1

```
void main(void )
{
  …
  for (;;) {
    OSSched();

    if ( !OSAnyEligibleTasks() ) {
      /* do alternative background process */
      #asm
      #include "mystuff.asm"
      #endasm
    }
  }
}
```

Example #2

```
void main(void )
{
  …
  for (;;) {
    OSSched();

    if ( !OSAnyEligibleTasks() )
      DoWhileIdling();
  }
}
```

OScTcbExt0|1|2|3|4|5, OStcbExt0|1|2|3|4|5(): Return a Tcb Extension

Type:	Macro
Declaration:	`OScTcbExt0|1|2|3|4|5, OStcbExt0|1|2|3|4|5(tcbP)`
Callable from:	`OScTcbExt0|1|2|3|4|5` should only be called from the task level. `OStcbExt0|1|2|3|4|5()` can be called from anywhere.
Contained in:	`salvo.h`
Enabled by:	`OSENABLE_TCBEXT0|1|2|3|4|5`
Affected by:	–
Description:	`OScTcbExt0|1|2|3|4|5` returns the specified tcb extension of the current task. `OStcbExt0|1|2|3|4|5` returns the specified tcb extension of the specified task.
Parameters:	–
Returns:	Tcb extension.
Stack Usage:	0.

Notes

These macros are used to obtain the desired tcb extension from the task's tcb.

See Also

`OSENABLE_TCBEXT0|1|2|3|4|5`, `OSTYPE_TCBEXT0|1|2|3|4|5`

Example

```
void CommTask ( void )
{
  /* ascertain mode at startup */
  switch ( OScTcbExt3 ) {

    case SW_HANDSHAKING:
      for (;;) {
        /* do comms w/ XON/XOFF */
        OpenSWUART();
        …
        OS_Yield(label1);
      }
      break;

    case HW_HANDSHAKING:
      for (;;) {
        /* do comms w/ DTR & CTS */
        OpenHWUART();
        …
        OS_Yield(label2);
      }
      break;

    default:
      break;
  }
}

main()
{
  …
  /* we want hardware handshaking … */
  OSCreateTask(CommTask, OSTCBP(7), 5);
  OStcbExt3(OSTCBP(7)) = HW_HANDSHAKING;
  …
  for(;;)
    OSSched();
}
```

OSDi(), OSEi(): Control Interrupts

Type:	Macro
Declaration:	`OSDi()`, `OSEi()`
Callable from:	Anywhere
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	–
Description:	Disable or enable interrupts, respectively.
Parameters:	–
Returns:	n/a
Stack Usage:	0, unless defined otherwise.

Notes

These macros are usually the same as the compiler's native `di()` / `ei()` or `disable()` / `enable()` pair of interrupt-control macros. They exist primarily so that sample programs can all use the same functions to control interrupts.

If you need to manage interrupts globally in your application, you may find it simpler to use Salvo's built-in interrupt control than to create your own. If `OSDi()` and `OSEi()` do not suit your purposes, you can redefine them in your `salvocfg.h`.

If you have any questions concerning how these macros are implemented for your compiler and target processor, look in `salvo.h` and `portXyz.h` for more information.

See Also

`OSENABLE_INTERRUPT_HOOKS`, `OSDisableIntsHook()`, `OSEnableIntsHook()`

Example

```
…
OSDi();
/* critical section of user code.              */
…
OSEi();
…
```

OSProtect(), OSUnprotect(): Protect Services Against Corruption by ISR

Type:	Macro
Declaration:	`OSProtect(), OSUnprotect()`
Callable from:	Background
Contained in:	`portXyz.h`
Enabled by:	–
Affected by:	–
Description:	Disable or enable interrupts, respectively, if such control is required on given target.
Parameters:	–
Returns:	n/a
Stack Usage:	0, unless defined otherwise.

Notes

When compiling for a target that does not have a software stack, certain steps must be taken to protect servicse with multiple callgraphs. By calling `OSProtect()` immediately before each such service, and `OSUnprotect()` immediately thereafter, the service is protected against any corruption that might occur if an interrupt that calls the service were to occur simultaneously.

These macros are empty for all targets whose compilers pass parameters on a stack. To ensure cross-platform compatibility, *all* Salvo applications should use `OSProtect()` and `OSUnprotect()` as specified, even if these macros are empty for a particular compiler.

Warning Because a stackless compiler may overlay the local / parameter areas of one or more services with multiple callgraphs, `OSProtect()` and `OSUnprotect()` should be used around *every* service whose `OSCALL_XYZ` is set to `OSFROM_ANYWHERE`.

In the example below, `OSSignalBinSem()` is called from mainline code and from within an ISR. Therefore `OSProtect()` and `OSUnprotect()` are required in the mainline code.

See Also

`OSCALL_OSXYZ`, `OSFROM_ANYWHERE`, `OSDi()`, `OSEi()`

Example

```
void TestCode( void )
{
    …
    if ( PutTx1Buff(data)) {
        OSProtect();
        OSSignalBinSem(BINSEM_TXBUFF_P);
        OSUnprotect();
    }
    …
}

void interrupt ISR( void )
{
    …
    if ( txState == TXSTATE_DONE ) {
        txState = TXSTATE_IDLE;
        OSSignalBinSem(BINSEM_TXDONE_P);
    }
    …
}
```

OSTimedOut(): Check for Timeout

Type:	Macro
Declaration:	`OSTimedOut()`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	`OSENABLE_TIMEOUTS`
Affected by:	–
Description:	Detect if the current task timed out waiting for an event.
Parameters:	–
Returns:	`TRUE` if a timeout occurred, `FALSE` otherwise.
Stack Usage:	0

Notes

By specifying a non-zero timeout in `OS_WaitBinSem()`, `OS_WaitMsg()`, `OS_WaitMsgQ()` or `OS_WaitSem()`, you can control program execution in the case where an event does not occur within a specified number of system ticks. This is very useful in handling errors that may result from expected events failing to occur.

In the example below, a bidirectional communications channel is used to send commands and receives a response (acknowledgments) for each command sent. A new command can be sent only after the acknowledgment for the previous command has been received. By specifying a response timeout (`RSP_TIMEOUT`) that's larger than the expected time for the receiver to respond to a command, `TaskTx()` can conditionally wait for the response instead of waiting indefinitely if the acknowledgment never arrives.

When a timeout occurs, a task's execution resumes where it was originally waiting for the event, and the Salvo function `OSTimedOut()` returns `TRUE` until the task context-switches back to the scheduler. `TaskTx()` checks to see if a timeout occurred after it acquires the message. If a timeout occurred, the message pointer is invalid.

See Also

`OS_WaitBinSem()`, `OS_WaitMsg()`, `OS_WaitMsgQ()`, `OS_WaitSem()`

Example

```
void TaskTx( void )
{
  static OStypeMsgP msgP;

  /* No cmds have been sent yet, so no         */
  /*  responses have been received.            */
  OSCreateMsg(MSG_RSP_RCVD_P, (OStypeMsgP) 0);

  for (;;) {
    /* send command to receiver.               */
    …

    /* wait here until response has been       */
    /*  received for the command we sent.      */
    /* if we timed out, reset the expected     */
    /*  response, STOP, clear the buffer and   */
    /*  tell the user.                         */
    OS_WaitMsg(MSG_RSP_RCVD_P, &msgP, RSP_TIMEOUT,
      TaskTx1);

    if ( OSTimedOut() ) {
      FlushCmdInterpreter();
      setSTOP();
      txBuff[0] = 0;
      FlashMsg(&msgBadComms);
    }

    /* continue processing outgoing commands.  */
    …
  }
}
```

OSVersion(), OSVERSION: Return Version as Integer

Type:	Macro
Declaration:	`OSVersion(), OSVERSION`
Callable from:	Anywhere
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	–
Description:	Returns the version number.
Parameters:	–
Returns:	Returns the version number as an unsigned integer.
Stack Usage:	0

Notes

Salvo uses three version number fields: `OSVER_MAJOR`, `OSVER_MINOR` and `OSVER_SUBMINOR`. Each field is a numeric integer constant. They are combined into a single symbol, `OSVERSION`, in the following manner:

```
OSVERSION = OSVER_MAJOR   * 100
          + OSVER_MINOR   *  10
          + OSVER_SUBMINOR
```

Therefore in v3.0.0, `OSVERSION` equals 300.

`OSVersion()` is identical to `OSVERSION`.

Example

```
printf("Salvo version: %d (v%c.%c.%c)\n",
  '0' + OSVER_MAJOR,
  '0' + OSVER_MINOR,
  '0' + OSVER_SUBMINOR,
  OSVersion());
```

User Macros

This section describes the Salvo user macros that you will use to build your multitasking application.

The macros are described below.

_OSLabel(): Define Label for Context Switch

Type:	Macro
Declaration:	`_OSLabel( labelname )`
Callable from:	n/a
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	`OSCOMPILER`
Description:	Creates a label for the compiler to reference for each context switch.
Parameters:	`labelname`: a unique name for a label associated with a particular context switch.
Returns:	n/a
Stack Usage:	n/a

Notes

Most compilers for use with Salvo require a unique name for the label associated with each context-switching user service. This macro creates the required label in the particular form necessary for the compiler in use.

By using this macro you can compile your application with different compilers without having to write your own compiler-dependent label declarations.

See Also

`OSCOMPILER`

Example

```
_OSLabel(Here)
_OSLabel(There)
_OSLabel(Everywhere)

void MyFnName( void )
{
  for (;;) {
    OS_Yield(Here);
    …
    OS_Delay(20, There);
    …
    OS_WaitBinSem(SEM_3, OSNO_TIMEOUT,
      Everywhere);
  }
}
```

OSECBP(), OSEFCBP(),OSMQCBP(), OSTCBP(): Return a Control Block Pointer

Type:	Macro
Declaration:	`OSECBP( index )` `OSEFCBP( index )` `OSMQCBP( index )` `OSTCBP( index )`
Callable from:	n/a
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	–
Description:	Shorthand for pointer to specified control block.
Parameters:	`index`: an index from `1` to `OSEVENTS`, `1` to `OSEVENT_FLAGS`, `1` to `OSMESSAGE_QUEUES` or `1` to `OSTASKS`, respectively.
Returns:	pointer to (i.e. address of) desired event, message queue or task control block, respectively.
Stack Usage:	n/a

Notes

RAM memory for control blocks is allocated at compile time using the `OSEVENTS`, `OSEVENT_FLAGS`, `OSMESSAGE_QUEUES` and `OSTASKS` configuration options. Instead of obtaining the compile-time address of a particular event, event flag, message queue or task control block by using

```
&OSecbArea[i-1]
&OsefcbArea[i-1]
&OSmqcbArea[i-1]
&OStcbArea[i-1]
```

you can and *should* use these macros.

Example

```
#define TASK1_P      OSTCBP(1)
#define TASK2_P      OSTCBP(2)
#define SEM1_P       OSECBP(1)

…
OSCreateTask(Task1, TASK1_P, 7);
…
OSCreateSem(SEM1_P, 14);
…
```

User-Defined Services

OSDisableIntsHook(), OSEnableIntsHook(): Interrupt-control Hooks

Type:	Function
Declaration:	`void OSDisableIntsHook( void )` `void OSEnableIntsHook( void )`
Called from:	`OSDi()` and `OSEi()`
Contained in:	`salvo.h` if left undefined, otherwise in user source code.
Enabled by:	`OSENABLE_INTERRUPT_HOOKS`
Affected by:	–
Description:	User-defined.
Parameters:	–
Returns:	–
Stack Usage:	Dependent on user definition.

Notes

You may find it useful or necessary to perform certain operations coincident with Salvo's disabling and (re-)enabling of interrupts during critical sections of code.

If these functions are enabled via `OSENABLE_INTERRUPT_HOOKS`, `OSDisableIntsHook()` is called *immediately after disabling interrupts*, and `OSEnableIntsHook()` is called *immediately before (re-)enabling interrupts*. Therefore each function is called with interrupts disabled.

By default, these functions are undefined.

In the example below, two separate counters, `diCounter` and `eiCounter`, are used to count the number of times that Salvo disables and (re-)enables interrupts, respectively.

See Also

`OSDi()`, `OSEi()`

Example

```
unsigned long int diCounter, eiCounter;
…
void OSDisableIntsHook( void )
{
  diCounter++;
}

void OSEnableIntsHook( void )
{
  eiCounter++;
}
```

OSIdlingHook(): Idle Function Hook

Type:	Function
Declaration:	`void OSIdlingHook( void )`
Called from:	`OSSched()`
Contained in:	User source code, called from `sched.c`.
Enabled by:	`OSENABLE_IDLING_HOOK`
Affected by:	–
Description:	User-defined.
Parameters:	–
Returns:	–
Stack Usage:	Dependent on user definition.

Notes

Salvo's scheduler normally runs in a tight loop when no tasks are eligible to run, i.e. when it is idling. By defining an idle function and setting `OSENABLE_IDLING_HOOK` to `TRUE`, you can do something useful while the system is idling. Your idle function should be short and fast, as time spent in it delays the operation of the scheduler.

By default, `OSIdlingHook()` is undefined.

In the example below, the least significant bit on an output port is toggled whenever there are no eligible or running tasks.

Example

```
void OSIdlingHook( void )
{
  PORTB ^= 0x01;
}
```

Return Codes

Many Salvo user services have return codes to indicate whether or not they were called successfully. They are listed below. See the individual user service descriptions for more information on return codes.

`OSNOERR`:	No error.
`OSERR`:	An error was encountered while executing the user service.
`OSERR_TASK_BAD_P`:	An invalid pointer was passed to the user service.
`OSERR_EVENT_NA`:	The specified event was not available
`OSERR_EVENT_FULL`:	The specified event (e.g. message) is already full.
`OERR_EVENT_CB-_UNINIT`:	The specified control block (e.g. for message queues or event flags) has not yet been initialized.
`OSERR_TIMEOUT`:	The current task has timed out while waiting for an event.

Table 6: Return Codes

Salvo Defined Types

The following types are defined for use with Salvo user services. Because the types are affected by configuration options, *when interfacing to Salvo user services you should always declare variables with these defined types*. Failing to do so is likely to result in unpredictable behavior.

Salvo has two classes of predefined types: those where the memory (RAM) location of the object is not specified (*normal*, `OStypeXyz`), and those where the location is explicitly specified (*qualified*, `OSgltypeXyz`). The need for both types arises on those processors with banked RAM. If your target processor has a single linear RAM space, the two types are identical. When in doubt, use the qualified type if one exists.

The normal types are used in the Salvo source code when declaring auto variables, parameters and function return values. You can also use the normal types when declaring your own local variables (e.g.

message pointers of type `OStypeMsgP`), and when typecasting (e.g. `OSSignalMsg(MSGP, (OStypeMsgP) &array[2])`);

The qualified types are used to declare Salvo's global variables, and are also provided so that you can properly declare your own global variables for Salvo, e.g. message queues – `OSgltypeMsgQP MsgQBuff[SIZEOF_MSGQ]`.

Tip Refer to the Salvo source code for examples of when to use normal or qualified Salvo types.

The normal types are:

`OStypeBinSem`:	binary semaphore: `OStypeBoolean`
`OStypeBitField`:	size of bit fields in structures: `int` or `char`, depending on `OSUSE_CHAR_SIZED_BITFIELDS`
`OStypeBoolean`:	Boolean: `FALSE` (0) or `TRUE` (non-zero)
`OStypeCount`:	counter: `OStypeInt8u/16u/32u`, depending on `OSBYTES_OF_COUNTS`
`OStypeDelay`:	delay: `OStypeInt8u/16u/32u`, depending on `OSBYTES_OF_DELAYS`
`OStypeDepth`:	stack depth counter: `OStypeInt8u`
`OStypeEcb:`	event control block: structure
`OStypeEfcb:`	event flag control block: structure
`OStypeEFlag`:	event flag: `OStypeInt8u/16u/32u`, depending on configuration
`OStypeErr`:	function return code or error / warning / timeout counter: `OStypeInt8u`
`OStypeEType`:	event type: `OStypeInt8u`
`OStypeID:`	object ID: `OStypeInt8u`
`OStypeInt8u`:	integer: 8-bit, unsigned
`OStypeInt16u`:	integer: 16-bit, unsigned
`OStypeInt32u`:	integer: 32-bit, unsigned
`OStypeInterval`:	interval: `OStypeInt8/16/32`, depending on `OSBYTES_OF_DELAYS`
`OStypeMqcb:`	message queue control block: structure
`OStypeMsg`:	message: `void` or `const`, depending on `OSBIG_MESSAGE_POINTERS`
`OStypeMsgQSize`:	number of messages in a message queue: `OStypeInt8u`
`OStypeOption:`	generic option: `OStypeInt8u`

OStypePrio:	task priority: OStypeInt8u, values from 0 to 15 are defined
OStypePS:	timer prescalar: OStypeInt8u/16u/32u, depending on configuration
OStypeSem:	semaphore: OStypeInt8u or OStypeInt16u, depending on configuration
OStypeState:	task state: OStypeInt8u, values from 0 to 7 are defined
OStypeStatus:	task status: bitfields of type OStypeInt8u for a task's running bit, state and priority
OStypeTcb:	task control block: structure
OStypeTcbExt:	tcb extension: void *, user-(re-)definable
OStypeTick:	timer ticks: OStypeInt8u/16u/32u, depending on configuration
OStypeTS:	timestamp: OStypeInt8u/16u/32u, depending on configuration of OSBYTES_OF_DELAYS

Table 7: Normal Types

The normal pointer types are:

OStypeCharEcbP:	pointer to banked (OSLOC_ECB) char
OStypeCharTcbP:	pointer to banked (OSLOC_TCB) char
OStypeEcbP:	pointer to banked (OSLOC_ECB) event control block
OStypeEfcbP:	pointer to banked (OSLOC_EFCB) event flag control block
OStypeMqcbP:	pointer to banked (OSLOC_MQCB) message queue control block
OStypeMsgP:	pointer to message
OStypeMsgPP:	pointer to pointer to message
OStypeMsgQPP:	pointer to banked (OSLOC_MSGQ) pointer to message
OStypeTcbP:	pointer to banked (OSLOC_TCB) task control block
OStypeTcbPP:	pointer to banked (OSLOC_ECB) pointer to banked (OSLOC_TCB) task control block
OStypeTFP:	pointer to (task) function

Table 8: Normal Pointer Types

The qualified types are:

OSgltypeCount:	qualified OStypeCount: banked (OSLOC_COUNT) counter
OSgltypeDepth:	qualified OStypeDepth: banked (OSLOC_DEPTH) stack depth counter
OSgltypeEcb:	qualified OStypeEcb: banked (OSLOC_ECB) event control block
OSgltypeEfcb:	qualified OStypeEfcb: banked (OSLOC_EFCB) event flag control block
OSgltypeErr:	qualified OStypeErr: banked (OSLOC_ERR) error counter
OSgltypeGlStat:	qualified OStypeGlStat: banked (OSLOC_GLSTAT) global status bits
OSgltypeLogMsg:	qualified char: banked (OSLOC_LOGMSG) log message character or string
OSgltypeMqcb:	qualified OStypeMqcb: banked (OSLOC_MQCB) message queue control block
OSgltypePS:	qualified OStypePS: banked (OSLOC_PS) timer prescalar
OSgltypeTcb:	qualified OStypeTcb: banked (OSLOC_TCB) task control block
OSgltypeTick:	qualified OStypeTick: banked (OSLOC_TICK) system ticks

Table 9: Qualified Types

The qualified pointer types are:

OSgltypeCTcbP:	qualified OStypeTcbP: banked (OSLOC_CTCB) pointer to banked task control block
OSgltypeEcbP:	qualified OStypeEcbP: banked (OSLOC_ECB) pointer to banked event control block
OSgltypeMsgQP:	qualified OStypeMsgP: banked (OSLOC_MSGQ) pointer to message
OSgltypeSigQP:	qualified OStypeTcbP: banked (OSLOC_SIGQ) pointer to banked task control block
OSgltypeTcbP:	qualified OStypeTcbP: banked (OSLOC_ECB) pointer to banked task control block

Table 10: Qualified Pointer Types

Note When declaring *pointers* using predefined Salvo pointer types on targets that have banked RAM, always declare each pointer on its own, like this:

```
OStypeMsgP msgP1;
OStypeMsgP msgP2;
```

Failing to do so (i.e. declaring multiple pointers by comma-delimiting them on one line) will result in an improper declaration.

Salvo Variables

Salvo's global variables (declared in `mem.c`) are listed below. The variable, the qualified type corresponding to the variable and a description of the variable are listed for each one. Advanced programmers may find it useful to read these variables during runtime or while debugging. In some development environments (e.g. Microchip MPLAB), these variable names will be available for symbolic debugging.

Warning Do not modify any of these variables during runtime – unpredictable results may occur.

`OScTcbP`	`OSgltypeCTcbP`	pointer to current task's task control block
`OSctxSws`	`OSgltypeCount`	context switch counter
`OSdelayQP`	`OSgltypeDelayQP`	pointer to delay queue
`OSecbArea[]`	`OSgltypeEcb`	event control block storage
`OSefcbArea[]`	`OSgltypeEfcb`	event flag control block storage
`OSeligQP`	`OSgltypeEligQP`	pointer to eligible queue
`OSerrs`	`OSgltypeErr`	runtime error counter
`OSglStat`	`OSgltypeGlStat`	global status bits
`OSidleCtxSws`	`OSgltypeCount`	idle function calls counter
`OSlogMsg[]`	`OSgltypeLogMsg`	log (debug) message string

OSmaxStkDepth	OSgltypeDepth	maximum stack depth achieved by Salvo functions
OSmqcbArea[]	OSgltypeMqcb	message queue control block storage
OSrtnAddr	OSgltypeTFP	task's return / resume address
OSsigQinP, OSsigQoutP	OSgltypeSigQP	signaled event queue insert and removal pointers
OSstkDepth	OSgltypeDepth	current stack depth of Salvo function
OStcbArea[]	OSgltypeTcb	task control block storage
OStimerTicks	OSgltypeTick	system timer ticks counter
OStimerPS	OSgltypePS	runtime timer prescalar
OStimeouts	OSgltypeErr	runtime timeout counter
OSwarns	OSgltypeErr	runtime warning counter

Table 11: Salvo Variables

Salvo Source Code

The Salvo source code is organized into files that handle tasks, resources, queues, data structures, utility functions, the monitor, and the many `#defines` that are used to configure Salvo for a variety of applications.

You can always review the source code if the manual is unable to answer your question(s). Modifying the source code is not recommended, as your application may not run properly when compiled with a later release of Salvo. Where applicable, user `#defines` and hooks for user functions are provided so that you can use Salvo in conjunction with features that are not yet supported in the current release.

Salvo's source (`*.h` and `*.c`) files are listed below.

```
salvo\inc\salvo.h
salvo\src\array.c
```

```
salvo\src\binsem.c
salvo\src\binsem2.c
salvo\src\chk.c
salvo\src\debug.c
salvo\src\delay.c
salvo\src\delay2.c
salvo\src\delay3.c
salvo\src\destroy.c
salvo\src\eflag.c
salvo\src\eflag2.c
salvo\src\eid.c
salvo\src\event.c
salvo\src\idle.c
salvo\src\init.c
salvo\src\initecb.c
salvo\src\inittask.c
salvo\src\inittcb.c
salvo\src\license.c
salvo\src\mem.c
salvo\src\msg.c
salvo\src\msg2.c
salvo\src\msgq.c
salvo\src\msgq2.c
salvo\src\msgq3.c
salvo\src\prio.c
salvo\src\prio2.c
salvo\src\qdel.c
salvo\src\qins.c
salvo\src\rpt.c
salvo\src\sched.c
salvo\src\sem.c
salvo\src\sem2.c
salvo\src\start.c
salvo\src\stop.c
salvo\src\task.c
salvo\src\task2.c
salvo\src\task3.c
salvo\src\task4.c
salvo\src\task5.c
salvo\src\task6.c
salvo\src\task7.c
salvo\src\tick.c
salvo\src\tid.c
salvo\src\timer.c
salvo\src\util.c
salvo\src\ver.c
```

Listing 37: Source Code Files

The main compiler-specific header and source files are listed below.

```
salvo\inc\portgcc.h
salvo\inc\portiarc.h
salvo\inc\portmcc.h
```

```
salvo\inc\portmixc.h
salvo\inc\portmwcw.h
salvo\inc\portpicc.h
salvo\inc\portv8c.h
salvo\src\portaq430.asm
salvo\src\port8051.c
salvo\src\portpic18.c
salvo\src\portv8c.c
salvo\src\portiar430.s43
```

Listing 38: Target-specific Source and Header Files

Additional configuration-specific files are listed below.

```
salvo\inc\salvolvl.h
salvo\inc\salvomcg.h
salvo\inc\salvoscg.h
```

The user-configurable compiler-specific file is listed below.

```
salvo\inc\user\portuser.h
```

A complete `salvocfg.h` file with all configuration options set to their default values is listed below.

```
salvo\inc\user\salvocfg.h
```

The `salvocfg.h` file used to generate the freeware libraries, and also used when linking to them via `OSUSE_LIBRARY`, is listed below.

```
salvo\inc\salvolib.h
```

Note Salvo source code uses tab settings of 4, i.e. tabs are equivalent to 4 spaces.

Locations of Salvo Functions

Below is a list of each Salvo function (including user services and certain internal functions called by user services, shown in *italics*) and the source file in which it resides. This list is provided to assist source code users in resolving compile-time link errors due to the failure to include a particular Salvo source code file in their project.

Note Under certain configurations, those functions marked with an '*' may be macros or in-lined code instead of functions.

OSClrEFlag()*	eflag.c
OSCreateBinSem()*	binsem.c
OSCreateEFlag()*	eflag.c
OSCreateEvent()	event.c
OSCreateMsg()*	msg.c
OSCreateMsgQ()*	msgq.c
OSCreateSem()*	sem.c
OSCreateTask()	inittask.c
OSDelay()	delay.c
OSDelDelayQ()	qdel.c
OSDelPrioA()	array.c
OSDelPrioQ()	qdel.c
OSDelTaskQ()	task7.c
OSDestroy()	destroy.c
OSDestroyTask()	task3.c
OSDispTcbP()	rpt.c
OSeID()	eid.c
OSGetPrio()*	prio2.c
OSGetPrioTask()	prio2.c
OSGetTicks()	ticks.c
OSGetState()	task.c
OSGetStateTask()	task5.c
OSGetTS()	delay2.c
OSInit()	init.c
OSInitEcb()	initecb.c
OSInitPrioTask()	inittask.c
OSInitTcb()	inittcb.c
OSInsDelayQ()	qins.c
*OSInsElig()**	qins.c
OSInsPrioA()	array.c
OSInsPrioQ()	qins.c
OSInsTaskQ()	task7.c
*OSLogErr()**	debug.c
*OSLogMsg()**	debug.c
*OSLogWarn()**	debug.c
OSMakeStr()	debug.c
OSMsgQEmpty()	msgq3.c
OSPrintEcb()	rpt.c
OSPrintEcbP()	rpt.c
OSPrintTcb()	rpt.c
OSPrintTcbP()	rpt.c
OSRestoreIntStat()	portpic18.c
OSRtnBinSem()	binsem2.c
OSRtnEFlag()	eflag2.c
OSRtnMsg()	msg2.c
OSRtnMsgQ()	msgq2.c
OSRtnSem()	sem2.c
OSRpt()	rpt.c
OSSaveIntStat	portpic18.c
OSSaveRtnAddr()	util.c
OSSched()*	sched.c
OSSetEFlag()*	eflag.c
OSSetPrio()	prio.c
OSSetPrioTask()	task6.c
OSSetTicks()	ticks.c

OSSetTS()	delay2.c
OSSignalBinSem()*	binsem.c
OSSignalEvent()	event.c
OSSignalMsg()*	msg.c
OSSignalMsgQ()*	msgq.c
OSSignalSem()*	sem.c
OSStartTask()	task.c
OSStop()	stop.c
OSStopTask()	task2.c
OSSyncTS()	delay3.c
OSTaskUsed()	task7.c
OSTaskRunning()	task4.c
OStID()	tid.c
OSTimer()*	timer.c
OSTryBinSem()	binsem.c
OSTryMsg()	msg.c
OSTryMsgQ()	msgq.c
OSTrySem()	sem.c
OSWaitEvent()	event.c

Listing 39: Location of Functions in Source Code

Abbreviations Used by Salvo

The following abbreviations are used throughout the Salvo source code:

address	addr
array	A
binary	bin
change	change, chg
check	chk
circular	circ
clear	clr
create	create
configuration	config
context	ctx
current	curr, c
delay	delay
delete	del
depth	depth
destroy	destroy
disable	dis
disable interrupt(s)	di
ecb pointer	ecbP
eligible	elig
enable	en
enable interrupt(s)	ei
enter	enter
event	event, e
event control block	ecb
event flag	eFlag
event flag control block	efcb
event type	eType

error	err
from	fm
global	gl
global type	gltype
identifier	ID
include guard	IG
initialize	init
insert	ins
length	len
local	l
location	loc
maximum	max
message	msg
message queue	msgQ
message queue control block	mqcb
minimum	min
not available	NA
number	num
operating system	OS
pointer	ptr, p
pointer to a pointer	pp
prescalar	PS
previous	prev
priority	prio
queue	Q
report	rpt
reset	rst
restore	rstr
return	rtn
save	save
scheduler	sched
semaphore	sem
set	set
signal	signal
stack	stk
status	stat
statistics	stats
string	str
switch	sw
synchronize	sync
task	task, t
task control block	tcb
task function pointer	tFP
tcb extension	tcbExt
tcb pointer	tcbP
tick	tick
timeout	timeout
timer	timer
timestamp	TS
toggle	tgl
utility	util
value	val
version	ver
wait(ing) (for)	wait, w
warning	warn

Listing 40: List of Abbreviations

Chapter 8 • Libraries

Library Types

Salvo ships with two types of precompiled libraries – *standard libraries* and *freeware libraries*. The standard libraries contain all of Salvo's basic functionality, configured for each supported compiler and target processor. The freeware libraries are identical to the standard libraries except for the relatively limited numbers of supported tasks and events.

Users of full Salvo distributions can create applications using the Salvo source files, the standard libraries, or a combination thereof.

Users of the freeware Salvo Lite distributions must use the freeware libraries when creating their applications. For functionality and flexibility greater than that provided by the freeware libraries, you'll need to purchase Salvo for full access to the Salvo source code, the standard libraries and all the configuration options.

Every Salvo distribution contains the associated freeware libraries. Only Salvo full-version distributions contain the associated standard libraries.

Libraries for Different Environments

The various Salvo distributions contain libraries for two different kinds of compilers – *native* and *non-native* compilers.

Native Compilers

By native compilers we mean compilers that generate output (usually in `.hex` format) for a specific embedded target. You would use a native compiler to create a Salvo application for a real product. Native compilers are usually *cross-compilers*, i.e. they run on one machine architecture (usually x86-based PCs) and generate code for another (e.g. Microchip PICmicro).

Non-native Compilers

By non-native compilers we mean compilers that generate code for another target altogether (usually an x86-based PC). Salvo's support for these "pure" compilers[102] is intended to facilitate cross-platform development of Salvo applications for embedded targets. Users can build C console applications and test, run, and debug them on their main development machine (e.g. a PC) before building the same application for the intended embedded target (e.g. a PICmicro MCU). The editing and debugging features available on PCs are powerful tools that can aid in project management, testing and debugging.

If you wish to develop your embedded application on the PC and then recompile your Salvo application for your embedded target, keep in mind that the non-native compilers generally lack any support for non-console-oriented subsystems that may exist on your embedded target. Therefore you will need to simulate things like serial I/O, A/D, D/A, interrupts, etc.

This "build on two, run on one" technique can be quite useful. For example, you could write, test and debug a Salvo application that passes floating-point data between two tasks via a message queue. The PC's enormous[103] resources (stdout buffers, memory, etc.), coupled with a good IDE, present an ideal environment for developing this sort of application. You could debug your application using `printf()` or the IDE's debugger. Once your application works on the PC – and as long as you've used C library functions that are also included in your target compiler's libraries – then buildin a Salvo application for the embedded target should be a snap!

Using the Libraries

In order to use a standard or freeware library, first you must identify the library appropriate for your compiler and target processor. When linking your application to a library, place the `OSUSE_LIBRARY` and `OSLIBRARY_XYZ` configuration options particular to your compiler into your `salvocfg.h`. These configuration options ensure that the same configuration options used to generate the chosen library will also be used in your source code.

---

102 As opposed to cross-compilers.

103 When compared to an embedded microcontroller.

For example, to use the full-featured standard library for HI-TECH PICC and the PIC16C77, your `salvocfg.h` file would contain only:

```
#define OSUSE_LIBRARY            TRUE
#define OSLIBRARY_TYPE           OSL
#define OSLIBRARY_CONFIG         OSA
#define OSLIBRARY_VARIANT        OSB
```

Listing 41: Example salvocfg.h for Use with Standard Library

and your project would link to the standard library `slp42Cab.lib`.

Please see *Chapter 5 • Configuration* for more information on these configuration options.

Note `OSCOMPILER` and `OSTARGET` are not included in the `salvocfg.h` file listed above. That's because in most cases Salvo can automatically detect the compiler in use and then set the target processor accordingly. This is done in the preprocessor via predefined symbols supplied by the compiler.

Overriding Default RAM Settings

Each library is compiled with default values for the number of objects (tasks, events, etc.). By setting configuration parameters in `salvocfg.h` and compiling `mem.c` separately it's possible to increase or decrease the RAM allocated to Salvo, and hence the number of objects in your application.

If the number of objects in your application is smaller than what the library is compiled for, or your application doesn't use certain objects (e.g. message queues) that have their own, dedicated control blocks, you can reduce Salvo's RAM usage. First, add the appropriate configuration options to `salvocfg.h`. Then compile `salvo\src\mem.c` with your new `salvocfg.h` and link the resulting object module (`mem.obj`) to your application. In this case, `mem.obj` must be linked *before* the library.

For example, to *reduce* the amount of RAM allocated to tasks in the above example to just two, your `salvocfg.h` file would contain:

```
#define OSUSE_LIBRARY            TRUE
#define OSLIBRARY_TYPE           OSL
#define OSLIBRARY_CONFIG         OSA
#define OSLIBRARY_VARIANT        OSB
#define OSTASKS                  2
```

Listing 42: Example salvocfg.h for Use with Standard Library and Reduced Number of Tasks

and you would link these three files in the order displayed:

```
main.obj, mem.obj, slp42Cab.lib
```

By adding the following two lines to your `salvocfg.h`:

```
#define OSEVENT_FLAGS            0
#define OSMESSAGE_QUEUES         0
```

Listing 43: Additional Lines in salvocfg.h for Reducing Memory Usage with Salvo Libraries

you can prevent any RAM from being allocated to event flag and message queue control blocks, respectively.

Caution This technique frees RAM for other uses in your application, and must be used with caution. If you reduce `OSTASKS` or `OSEVENTS` from their default values, you must ensure that you do not perform any Salvo services on tasks or events that are now "out of range." E.g. for libraries that support three tasks, if you reduce `OSTASKS` to 2 as outlined above, you must not call `OSCreateTask(TaskName, OSTCBP(3), prio)`. If any of your own variables are located in RAM immediately after the tcbs, they will be overwritten with the call to `OSCreateTask()`.

The number of objects in an application can also be *increased*. This is only possible with the standard libraries – the preset limits in the freeware libraries cannot be overridden.

Note Illegal or incorrect values for the number of objects in an application that uses a library will usually be flagged by the compiler as an error.

Library Functionality

By linking your application to the appropriate library, you can use as few or as many of Salvo's user services as you like. Each library supports up to some number of tasks and events. Most libraries re-

quire that you create your own `OSIdlingHook()` function, which will be called automatically by the built-in idle function.

Note Because of the enormous number of possible configurations, the standard and freeware libraries support most, but not all, of Salvo's functionality. Each library is compiled with a particular set of configuration options. See the library-specific details (below) or `salvo\inc\salvolib.h` for more information.

Warning Do not edit `salvo\inc\salvolib.h`. Doing so may cause problems when compiling and/or linking your application to the freeware libraries.

Types

The library *type* is specified using the `OSLIBRARY_TYPE` configuration option in `salvocfg.h`.

The library types, shown in Table 12, are self-explanatory.

type code	description
`f / OSF:`	*F*reeware library. Number of tasks, events, etc. is restricted.[104]
`l / OSL:`	Standard *l*ibrary. Number of tasks, events, etc. is limited only by available RAM.

Table 12: Type Codes for Salvo Libraries

Note Users of the full Salvo distribution should always use the standard libraries, as they are slightly smaller than the freeware libraries.

Memory Models

Where applicable, Salvo libraries are compiled for different memory models. There is no configuration option for specifying the memory model.

104 Most freeware libraries are compiled with `OSSET_LIMITS` set to `TRUE`.

Options

Where applicable, Salvo libraries are compiled with different options. There is no configuration option for specifying the option.

Global Variables

Salvo uses a variety of objects for internal housekeeping. Where applicable, the `OSLIBRARY_GLOBALS` configuration option in `salvocfg.h` is used to specify the storage type for these global variables. The configuration codes vary by compiler.

Configurations

The library *configuration* is specified using the `OSLIBRARY_CONFIG` configuration option in `salvocfg.h`.

The library configurations, shown in Table 13, indicate which services are included in the library specified. Use the library that includes the minimum functionality that your application requires. For example, don't use an `a`-series library unless your application requires both delay (e.g. `OS_Delay()`) and event (e.g. `OSSignalSem()`) services.

configuration code	description
`a / OSA:`	Library supports multitasking with delay and event services – *a*ll default functionality is included.
`d / OSD:`	Library supports multitasking with *d*elay services only – event services are not supported.
`e / OSE:`	Library supports multitasking with *e*vent services only – delay services are not supported.
`m / OSM:`	Library supports *m*ultitasking only – delay and event services are not supported.
`t / OST:`	Library supports multitasking with delay and event services. Tasks can wait on events with a *t*imeout.

Table 13: Configuration Codes for Salvo Libraries

Note Using a library that's been created with support for services you don't use will have an impact on your application's ROM and RAM requirements.

Table 14 shows the differences among the library configurations.

configuration	a	d	e	m	t
Delay size:	8 bits	8 bits	-	-	8 bits
Event services:	enabled	disabled	enabled	disabled	enabled
Idling function:	enabled	enabled	enabled	disabled	enabled
Task priorities:	enabled	enabled	enabled	disabled	enabled
Timeouts:	disabled	disabled	disabled	disabled	enabled

Table 14: Features Common to all Salvo Library Configurations

Variants

The library *variant* is specified using the `OSLIBRARY_VARIANT` configuration option in `salvocfg.h`.

A variety of different compilers are certified for use with Salvo. Some compilers use the target processor's stack or registers to pass parameters and store auto variables – this is true for all compilers for x86 targets. *There are no library variants for these conventional compilers.*

Other compilers certified for use with Salvo maintain parameters and auto variables as static objects in dedicated RAM – this is the case for targets that do not have or use general-purpose stacks for parameter and auto variable storage. *The libraries for these compilers have variants.* The remainder of this section applies to the libraries for these compilers.

Some of Salvo's services can be called from within interrupts. Those services include:

- `OSGetPrioTask()`
- `OSGetStateTask()`
- `OSReadBinSem()`
- `OSReadEFlag()`
- `OSReadMsg()`
- `OSReadMsgQ()`
- `OSReadSem()`
- `OSMsgQEmpty()`
- `OSSignalBinSem()`
- `OSSignalMsg()`
- `OSSignalMsgQ()`
- `OSSignalSem()`
- `OSStartTask()`

Listing 44: Partial Listing of Services than can be called from Interrupts

If the target processor does not have a general-purpose stack, the Salvo source code must be properly configured via the appropriate configuration parameters. The library variants, shown in Table 15, are provided for those applications that call these services from within interrupts.

If your application does not call any of the services above from within interrupts, use the *b* variant. If you wish to these services exclusively from within interrupts, use the *f* variant. If you wish to do this from both inside and outside of interrupts, use the *a* variant. In each case, you must call the services that you use from the correct place in your application, or either the linker will generate an error or your application will fail during runtime.

variant code	description
`a / OSA:`	Applicable services can be called from *a*nywhere, i.e. from the foreground and the background, simultaneously.
`b / OSB:`	Applicable services may only be called from the *b*ackground (default).
`e / OSE:`	Applicable services may only be called from *e*ither the foreground or the background, but not both.
`f / OSF:`	Applicable services may only be called from the *f*oreground.
`- / OSNONE:`	Library has no variants.[105]

Table 15: Variant Codes for Salvo Libraries

See the `OSCALL_OSXYZ` configuration parameters for more information on calling Salvo services from interrupts.

105 A library may have no variants if the target processor does not support interrupts, or if the target processor has a conventional stack and the ability to save and restore the state of interrupts.

Library Reference

This section is organized by target and supported compiler(s).

Microchip PICmicro MCUs: HI-TECH PICC Compiler

The HI-TECH PICC C compiler is a Win32- and Linux-based C cross-compiler for use with Microchip baseline, mid-range and high-end PICmicro devices.[106] This compiler *does not* pass parameters on the (PICmicro's) stack. This compiler is commonly used with either the HI-TECH HPDPIC or HI-TIDE IDEs or the Microchip MPLAB IDE.

Compiler and Target Processor Configuration Options

This is done automatically through the `HI_TECH_C`, `_PIC12`, `_PIC14`, `_PIC16` and symbols defined by the HI-TECH PICC compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 52. It is similar to that used by HI-TECH for the standard PICC libraries.[107]

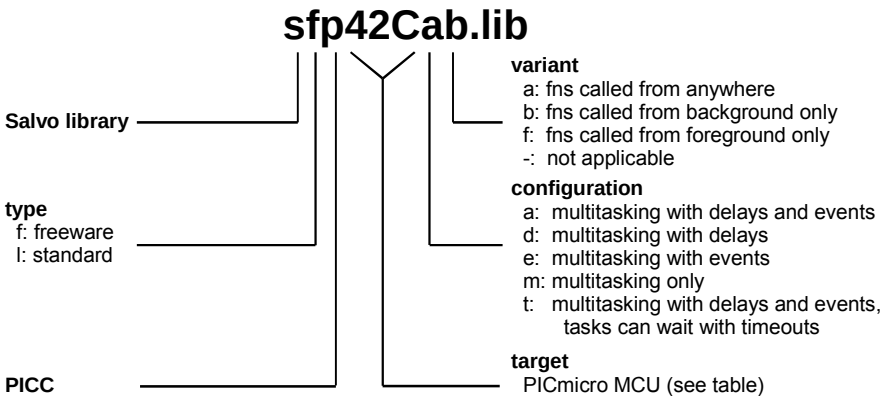


Figure 52: Salvo Library Nomenclature – HI-TECH PICC C Compiler

106 The freeware PICC-Lite compiler has the same features as PICC, but is limited to a small subset of the entire PICmicro line.

107 As of PICC v7.87.

Example salvocfg.h

The library type, configuration and variant must be specified when using a Salvo library for the HI-TECH PICC C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 52 is shown in Listing 45.

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_CONFIG        OSA
#define OSLIBRARY_VARIANT       OSB
```

Listing 45: Example salvocfg.h for Library Builds – HI-TECH PICC C Compiler

Build Settings

The Salvo libraries for the HI-TECH family of PICC C compilers are compiled with the configuration options listed in Table 16. All other configuration options that are not affected by the library's *configuration* (see below) are at their default values.[108] Differences between the PIC12 and other families are due to RAM constraints in the PIC12 series.

[108] For example, these particular libraries are compiled with `OSBIG_MESSAGE_POINTERS` set to `FALSE` (the default). Therefore the Salvo message pointers in PICmicro applications that use these libraries cannot point to ROM – they can only point to RAM.

configuration	value for PIC12	value for PIC16, PIC17
Tasks:	3	3
Events:	4	5
Event flags:[109]	1	1
Message queues:[110]	0	1
Delay sizes:	see Table 14	see Table 14
Idling function:	disabled	see Table 14
Task priorities:	disabled	see Table 14
Watchdog timer:	cleared in `OSSched()`	cleared in `OSSched()`
Salvo objects:[111,112]	`bank1 persistent`	`bank1 persistent`
System tick counter:	not available	available, 32 bits
Interrupt level:[113]	0	0

Table 16: Configuration Options Used to Build Salvo Libraries – HI-TECH PICC C Compiler

Note Because the `persistent` bank qualifier is used to build these libraries, `OSInit()` *must* be used in all applications that use these libraries. Without it, Salvo's variables will be uninitialized, with unpredictable results.

Note PIC12, PIC16 and PIC17 Salvo libraries are configured for 8-bit message pointers (PICC's default pointer type). These pointers can point to any area of banked RAM, *but they cannot point to ROM*, i.e. they cannot be used with, say, `const char` objects.

Processors

Each library is intended for one or more specific *processors*. Table 17 lists the correct library for each PICmicro® MCU.

109 Each event flag has RAM allocated to its own event flag control block.

110 Each message queue has RAM allocated to its own message queue control block.

111 PICmicro MCUs with only one bank of RAM have their Salvo variables located in RAM Bank 0.

112 By making Salvo's variables `persistent`, the PICC compiler is able to omit some initialization code and thus reduce ROM requirements.

113 Argument for PICC's `#pragma interrupt_level` for those services that can be called from within an ISR.

target code	processor(s)
200:	12C508(A), 12CE518, 16C52, 16C(R)54(A, B, C), 16HV540, 16C55(A)
210:	16C(R)56(A)
211:	12C(R)509(A, AF, AG), 12CE519
212:	16C505
222:	16C(R)57(A, B, C), 16C(R)58(A, B)
401:	16C554(A), 16C556(A), 16C558(A), 16C(R)62(A, B), 16C620, 16C621, 16C622, 16C(R)64(A), 16C712, 16C715, 16C716, 16C(R)72(A)
40a:	16C61, 16C71, 16C710, 16C711, 16CR83, 16C(R)84, 16F83, 16F84(A)
40b:	12C671, 12C672, 12CE673, 12CE674, 16C432, 16C433, 16C620A, 16C621A, 16C622A, 16C641, 16C661, 16CE623, 16CE624, 16CE625
40c:	16C717, 16C770, 16C781, 16C782, 16F627, 16F628, 16F870, 16F871, 16F872
411:	14000, 16C(R)63(A), 16C(R)65(A, B), 16C73(A, B), 16(L)C74(A, B)
412:	16F73, 16F74, 16F873(A), 16F874(A)
41b:	16C642, 16C662
41c:	16C771, 16C773, 16C774, 16C923, 16C924, 16C925
42c:	16C66, 16C67, 16C745, 16C76, 16C765, 16C77, 16F76, 16F77, 16F876(A), 16F877(A)
700:	17C42(A)
710:	17C42(A) with external ROM @ 7000-7FFFh[114]
704:	17C(R)43, 17C44, 17C752, 17C762
714:	17C756(A), 17C766
926:	16C926

Table 17: Processors for Salvo Libraries – HI-TECH PICC C Compiler

[114] This indicates to the code generator that external memory in addition to internal memory is present, and it will use (function) calls of a longer form. See the PICC User's Guide for more information on the `-ROM` command-line option.

Notes

There are 470 Salvo libraries for the HI-TECH PICC C compiler. (50 `p2xx`/PIC12, 300 `p4xx`/PIC16 and 120 `p7xx`/PIC17).

When using the `a`-variant libraries, each instance of an applicable service in use must be called from the foreground, i.e. from an interrupt. Also, PICC's `interrupt_level` pragma must be set to 0 and placed immediately ahead of the application's interrupt routine, like this:

```
#pragma interrupt_level 0
void interrupt IntVector( void )
{
  OSStartTask(TASK_P);
}
```

Listing 46: Setting the HI-TECH PICC interrupt_level pragma for an ISR when Using a-variant Libraries

Please see *AN-1 Using Salvo Freeware Libraries with the HI-TECH PICC Compiler* for specific information on using PICC compilers with the freeware libraries.

Microchip PICmicro MCUs: HI-TECH PICC-18 C Compiler

The HI-TECH PICC-18 C compiler is a Win32- and Linux-based C cross-compiler for use with Microchip enhanced PICmicro devices. This compiler *does not* pass parameters on the (PICmicro's) stack. This compiler is commonly used with either the HI-TECH HPDPIC or HI-TIDE IDEs or the Microchip MPLAB IDE.

Compiler and Target Processor Configuration Options

This is done automatically through the `HI_TECH_C` and `_PIC18` symbols defined by the HI-TECH PICC-18 C compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 52. It is similar to that used by HI-TECH for the standard PICC-18 libraries.[115]

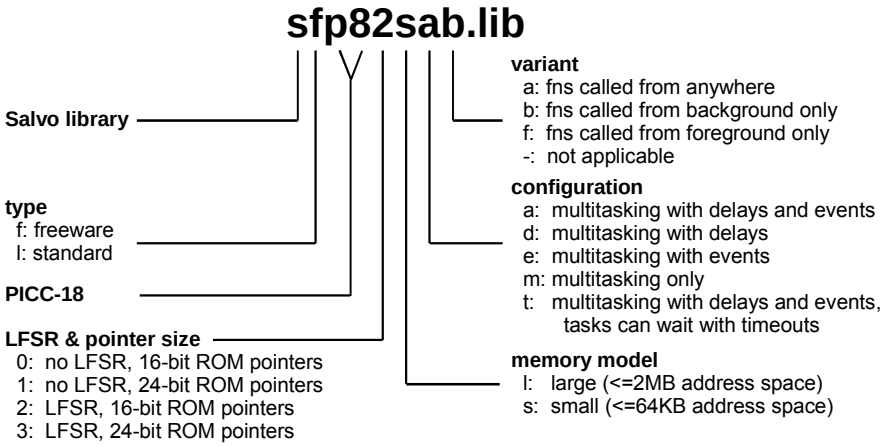


Figure 53: Salvo Library Nomenclature – HI-TECH PICC-18 C Compiler

Example salvocfg.h

The library type, configuration and variant must be specified when using a Salvo library for the HI-TECH PICC-18 C compiler. The

[115] As of PICC-18 v8.20.

appropriate `salvocfg.h` for the library illustrated in Figure 53 is shown in Listing 47.

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_CONFIG        OSA
#define OSLIBRARY_VARIANT       OSB
```

Listing 47: Example salvocfg.h for Library Builds – HI-TECH PICC-18 C Compiler

Build Settings

The Salvo libraries for the HI-TECH PICC-18 C compiler are compiled with the configuration options listed in Table 18. All other configuration options that are not affected by the library's *configuration* (see below) are at their default values.[116]

configuration	value
Tasks:	3
Events:	5
Event flags:[117]	1
Message queues:[118]	1
Delay sizes:	see Table 14
Idling function:	see Table 14
Task priorities:	see Table 14
Watchdog timer:	cleared in `OSSched()`
Salvo objects:[119]	`persistent`
System tick counter:	available, 32 bits
Interrupt level:[120]	0

Table 18: Configuration Options Used to Build Salvo Libraries – HI-TECH PICC-18 C Compiler

116 For example, these particular libraries are compiled with `OSBIG_MESSAGE_POINTERS` set to `FALSE` (the default). Therefore the Salvo message pointers in PICmicro applications that use these libraries cannot point to ROM – they can only point to RAM.

117 Each event flag has RAM allocated to its own event flag control block.

118 Each message queue has RAM allocated to its own message queue control block.

119 By making Salvo's variables `persistent`, the PICC-18 C compiler is able to omit some initialization code and thus reduce ROM requirements.

120 Argument for PICC-18's `#pragma interrupt_level` for those services that can be called from within an ISR.

Note Because the `persistent` bank qualifier is used to build these libraries, `OSInit()` *must* be used in all applications that use these libraries. Without it, Salvo's variables will be uninitialized, with unpredictable results.

Note PIC18 Salvo libraries are configured for 16-bit pointers (PICC-18's default pointer type), and therefore message pointers can point to RAM and ROM.

Processors

Each library is intended for one or more specific *processors*. Table 19 lists the correct library for each PIC18 PICmicro® MCU.

target code(s)	processor(s)
`80s, 80l, 81s, 81l:`	all PIC18 PICmicro® MCUs. Recommended only for 18C242, 18C252, 18C442, 18C452
`82s, 82l, 83s, 83l:`	all PIC18 PICmicro® MCUs *except* 18C242, 18C252, 18C442, 18C452. Recommended for all other PIC18 PICmicro® MCUs.

Table 19: Processors for Salvo Libraries – HI-TECH PICC-18 C Compiler

Note The `p80s`/`p80l`/`p81s`/`p81l` Salvo libraries are for those early PIC18 PICmicro® MCUs that do not support the `LFSR` instruction correctly[121] (e.g. 18C242, 18C252, 18C442, 18C452). *All* later devices in the family support this instruction correctly.

On targets that support `LFSR` correctly, using the `p82s`/`p82l`/`p83s`/`p83l` libraries will result in smaller and faster code.

Notes

There are 240 Salvo libraries for the HI-TECH PICC-18 C compiler.

When using the `a`-variant libraries, each instance of an applicable service in use must be called from the foreground, i.e. from an in-

[121] See Microchip's PIC18CXX2 Errata.

terrupt. Also, PICC-18's `interrupt_level` pragma must be set to 0 and placed immediately ahead of the application's interrupt routine, like this:

```
#pragma interrupt_level 0
void interrupt IntVector( void )
{
  OSStartTask(TASK_P);
}
```

Listing 48: Setting the HI-TECH PICC-18 interrupt_level pragma for an ISR when Using a-variant Libraries

Please see *AN-1 Using Salvo Freeware Libraries with the HI-TECH PICC Compiler* for specific information on using PICC compilers with the freeware libraries.

Microchip PIC18 PICmicro MCUs: IAR PIC18 C Compiler

The IAR PIC18 C compiler is a Win32- and Linux-based cross-compiler for use with Microchip PIC18 PICmicro MCUs. This compiler supports both static overlay and software stack code models, and uses the PIC18's hardware stack for call … return addresses. This compiler is commonly used within IAR's Embedded Workbench IDE and C-SPY debugger. It can also be used with Microchip's MPLAB IDE.

Compiler and Target Processor Configuration Options

This is done automatically through the `__IAR_SYSTEMS_ICC` and `__TID__` symbols defined by the IAR PIC18 C compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 54.[122]

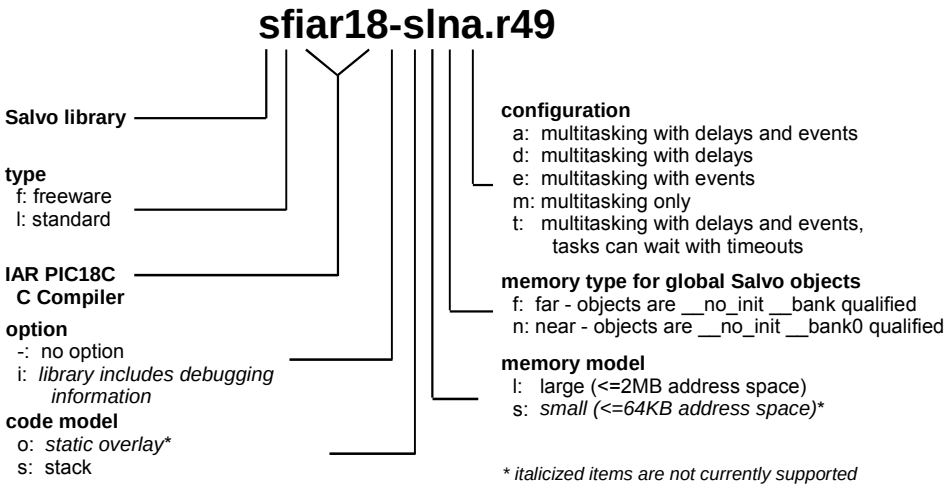


Figure 54: Salvo Library Nomenclature – IAR PIC18C C Compiler

[122] The PIC18 architecture uses 16-bit instructions and has a byte-addressible program memory space. Therefore 64KB is 32K instructions, and requires 16 addressing bits.

Example salvocfg.h

The library type, globals and configuration must be specified when using a Salvo library for the IAR PIC18 C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 54 is shown in Listing 50.

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_GLOBALS       OSN
#define OSLIBRARY_CONFIG        OSA
```

Listing 49 Example salvocfg.h for Library Builds – IAR PIC18C C Compiler

It is not necessary to indicate which code and memory models the library was built for (a mismatch will generate a link-time error with an obvious message). There's no need to specify the option code, either.

Build Settings

The Salvo libraries for the IAR PIC18 C compiler are compiled with the configuration options listed in Table 20. All other configuration options by the library's configuration (see below) are at their default values.

configuration	value
Tasks:	3
Events:	5
Event flags:	1
Message queues:	1
Delay sizes:	see Table 14
Idling function:	see Table 14
Task priorities:	see Table 14
Watchdog timer:	cleared in `OSSched()`
Salvo objects:	see Table 22
System tick counter:	available, 32 bits
Memory model:	see Table 21

Table 20: Configuration Options Used to Build Salvo Libraries – IAR PIC18C C Compiler

Currently the IAR PIC18 C compiler uses 24-bit ROM addressing – a small memory model is not yet supported. Therefore you must choose a Salvo library built with the large memory model. The memory model codes are listed in Table 21.

`l:`	*L*arge memory model. Program space is a maximum of 1M words (2MB) for PIC18 PICmicros with external ROM, and 32M words (64KB) for PIC18 PICmicros with internal ROM.
`s:`	*S*mall memory model. Program space is a maximum of 32K words (64KB).

Table 21: Memory Model Codes for Salvo Libraries – IAR PIC18 C Compiler

You can choose the memory type for Salvo's global objects in your application by choosing the appropriate library. `__bank0` type objects can be accessed the fastest, but consume precious RAM in the Access Bank. `__bank` type objects will be placed in banked RAM, which will result in slower accesses. The global object codes codes are listed in Table 23.

`f / OSF:`	Salvo objects are declared as type `__no_init __bank`, and will be located in banked RAM.
`n / OSN:`	Salvo objects are declared as type `__no_init __bank0`, and will be located in the first 128 bytes of internal RAM (i.e. in access RAM).

Table 22: Global Salvo Object Codes for Salvo Libraries – IAR PIC18 C Compiler

Processors

The PIC18's core architecture[123] is consistent throughout the entire family. Applications compiled for parts with external ROM may use the entire 21-bit address space, whereas those for internal-ROM-only parts are limited to a maximum program size of 32K instructions. Table 27 lists the allowed memory model codes for each PIC18 PICmicro MCU.

[123] I.e. the non-peripheral SFRs, like `TOSU|H|L`, `STKPTR`, `PCLATH|L`, `PCL`, `FSR0H|L`, etc.

memory model code	processor(s)
`l:`	18C242, 18C252, 18C442, 18C452, 18C658, 18C858, 18F010, 18F012, 18F020, 18F022, 18F122, 18F132, 18F242, 18F252, 18F258, 18F442, 18F452, 18F458, 18F6620, 18F6720, 18F8620, 18F8720, 18C601, 18C801
`s:`	*not currently supported*

Table 23: Processor Types for Salvo Libraries – IAR PIC18 C Compiler

Notes

There are 20 Salvo libraries for the IAR PIC18 C Compiler.

There are currently no Salvo libraries for use with the IAR PIC18 C compiler using the static overlay code model.

Microchip PIC18 PICmicro MCUs: MPLAB-C18 C Compiler

The Microchip MPLAB-C18 C compiler is a Win32-based cross-compiler for use with Microchip PIC18 PICmicro MCUs. This compiler passes parameters on a software stack, and uses the PIC18's hardware stack for call … return addresses. This compiler is commonly used within Microchip's MPLAB IDE.

Compiler and Target Processor Configuration Options

This is done automatically through the `__18CXX` symbol defined by the Microchip MPLAB-C18 C compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 55.[124]

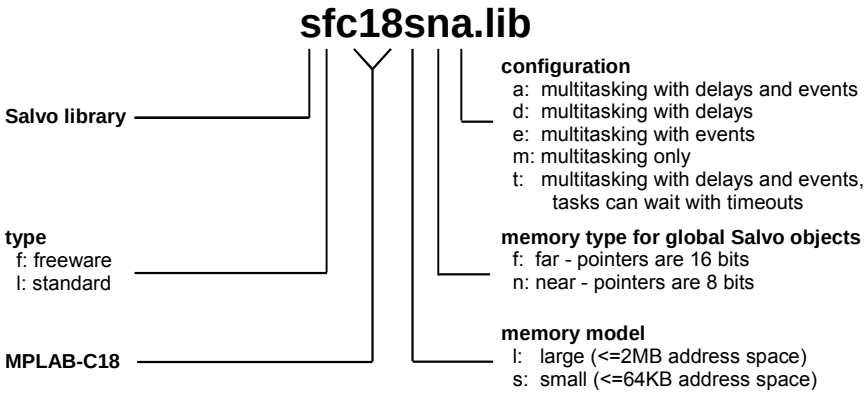


Figure 55: Salvo Library Nomenclature – Microchip MPLAB-C18 C Compiler

Example salvocfg.h

The library type, globals, configuration and variant must be specified when using a Salvo library for the Microchip MPLAB-C18 C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 55 is shown in Listing 50.

---

124 The PIC18 architecture uses 16-bit instructions and has a byte-addressible program memory space. Therefore 64KB is 32K instructions, and requires 16 addressing bits.

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_GLOBALS       OSN
#define OSLIBRARY_CONFIG        OSA
#define OSLIBRARY_VARIANT       OSNONE
```

Listing 50 Example salvocfg.h for Library Builds – Microchip MPLAB-C18 C Compiler

It is not necessary to indicate which memory model the library was built for (a mismatch will generate a link-time error with an obvious message).

Build Settings

The Salvo libraries for the Microchip MPLAB-C18 C compiler are compiled with the configuration options listed in Table 24. All other configuration options by the library's configuration (see below) are at their default values.

configuration	value
Tasks:	3
Events:	5
Event flags:	1
Message queues:	1
Delay sizes:	see Table 14
Idling function:	see Table 14
Task priorities:	see Table 14
Watchdog timer:	cleared in `OSSched()`
Salvo objects:	see Table 26
System tick counter:	available, 32 bits
Memory model:	see Table 25

Table 24: Configuration Options Used to Build Salvo Libraries – Microchip MPLAB-C18 C Compiler

Two memory models can be used when building applications with the Microchip MPLAB-C18 C compiler. You must use the Salvo library that matches the memory model of the rest of your application and your target processor. The memory model codes are listed in Table 25.

`l:`	*L*arge memory model. Program space is a maximum of 1M words (2MB). Only for PIC18 PICmicro MCUs with external ROM.
`s:`	*S*mall memory model. Program space is a maximum of 32K words (64KB).

Table 25: Memory Model Codes for Salvo Libraries – Microchip MPLAB-C18 C Compiler

You can choose the memory type for Salvo's global objects in your application by choosing the appropriate library. `near` type objects can be accessed the fastest, but consume precious RAM in the Access Bank. `far` type objects will be placed in banked RAM, which will result in slower accesses. The global object codes are listed in Table 26.

`f / OSF:`	Salvo objects are declared as type `far`, and will be located in banked RAM.
`n / OSN:`	Salvo objects are declared as type `near`, and will be located in the first 128 bytes of internal RAM (i.e. in access RAM).

Table 26: Global Salvo Object Codes for Salvo Libraries – Microchip MPLAB-C18 C Compiler

Processors

The PIC18's core architecture[125] is consistent throughout the entire family. Every Salvo library compiled under the small memory model can be used with any member of the PIC18 family. However, libraries for the Microchip MPLAB-C18 C compiler compiled under the large memory model can *only* be used with ROMless parts. Table 27 lists the allowed memory model codes for each PIC18 PICmicro MCU.

[125] I.e. the non-peripheral SFRs, like `TOSU|H|L`, `STKPTR`, `PCLATH|L`, `PCL`, `FSR0H|L`, etc.

memory model code	processor(s)
`l:`	18C601, 18C801
`s:`	18C242, 18C252, 18C442, 18C452, 18C658, 18C858, 18F010, 18F012, 18F020, 18F022, 18F122, 18F132, 18F242, 18F252, 18F258, 18F442, 18F452, 18F458, 18F6620, 18F6720, 18F8620, 18F8720, 18C601, 18C801

Table 27: Processor Types for Salvo Libraries – Microchip MPLAB-C18 C Compiler

Notes

There are 40 Salvo libraries for the Microchip MPLAB-C18 C compiler.

8051 Family: HI-TECH 8051C C Compiler

The HI-TECH 8051C C compiler is a DOS- and Linux-based cross-compiler for use with the 8051 and its derivatives. When using the large (memory) model, a conventional stack is employed for function return addresses, parameter passing, auto variables, etc. In the small and medium models, the stack is used only for function return addresses, and parameters and local variables are stored either in registers (if they fit), or in static storage. 8051C is commonly used either in the command-line mode or as part of the HPD51 IDE.

Compiler and Target Processor Configuration Options

This is done automatically through the `HI_TECH_C` and `i8051` symbols defined by the HI-TECH 8051C C compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 56.

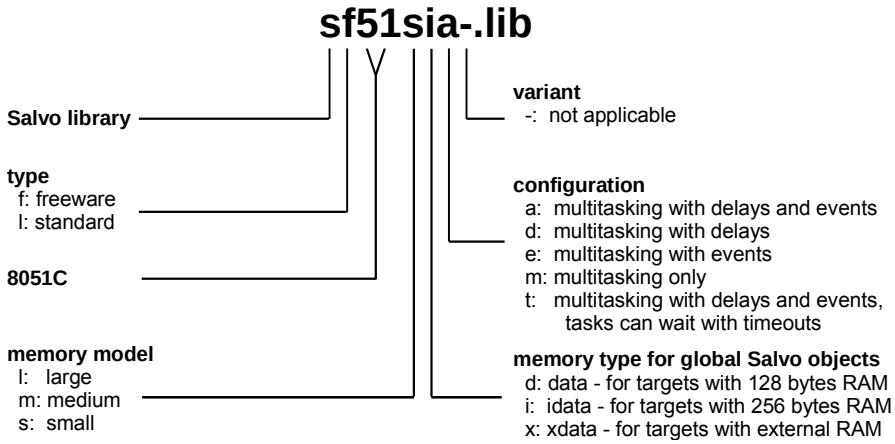


Figure 56: Salvo Library Nomenclature – HI-TECH 8051C C Compiler

Note Currently, the only memory types for global Salvo objects (which include pointers) in 8051C libraries is *xdata*.

Example salvocfg.h

The library type, globals, configuration and variant must be specified when using a Salvo 8051C library. The appropriate `salvocfg.h` for the library illustrated in Figure 56 is shown in Listing 51.

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_GLOBALS       OSI
#define OSLIBRARY_CONFIG        OSA
#define OSLIBRARY_VARIANT       OSNONE
```

Listing 51: Example salvocfg.h for Library Builds – HI-TECH 8051C C Compiler

It is not necessary to indicate which memory model the library was built for (a mismatch will generate a link-time error with an obvious message).

Build Settings

The Salvo libraries for the HI-TECH 8051C C compiler are compiled with the configuration options listed in Table 31. All other configuration options that are not affected by the library's *configuration* (see below) are at their default values.

configuration	value
Event flags:[126]	1
Events:	5
Message queues:[127]	1
Tasks:	3
Memory model:	see Table 29
Salvo objects:	see Table 30
System tick counter:	available, 32 bits
Watchdog timer:	not cleared
Task priorities:	see Table 14
Idling function:	see Table 14

Table 28: Configuration Options Used to Build Salvo Libraries – HI-TECH 8051C C Compiler

Various memory models can be used when building applications with the HI-TECH 8051C C compiler. You must use the Salvo li-

126 Each event flag has RAM allocated to its own event flag control block.

127 Each message queue has RAM allocated to its own message queue control block.

brary that matches the memory model of the rest of your application. The memory model codes are listed in Table 29.

`l:`	*L*arge memory model.
`m:`	*M*edium memory model.
`s:`	*S*mall memory model.

Table 29: Memory Model Codes for Salvo Libraries – HI-TECH 8051C C Compiler

You can choose the memory type for the Salvo objects in your application by choosing the appropriate library. `near` type objects can be accessed the fastest, but consume precious RAM on 8051s with only 128 bytes of RAM. `idata` type objects can be used on 8051s with 256 bytes of RAM. `far` type objects will be placed in external RAM, which will result in slower accesses. The memory type codes are listed in Table 30.

`d / OSD:`	Salvo objects are declared as type `near (data)`, and will be located in the first 128 bytes of internal RAM.
`I / OSI:`	Salvo objects are declared as type `idata`, and will be located in the first 256 bytes of internal RAM.
`x / OSX:`	Salvo objects are declared as type `far (xdata)`, and will be located in external RAM.

Table 30: Salvo Object Memory Type Codes for Salvo Libraries – HI-TECH 8051C C compiler

Notes

8051 Family: Keil Cx51 C Compiler

The Keil Cx51[128] C compiler is a Win32-based cross-compiler for use with the 8051 and its derivatives. This compiler *does not* pass parameters and auto variables on the stack unless a function is declared as reentrant. It is commonly used either in the command-line mode or as part of Keil's µVision IDE.

Compiler and Target Processor Configuration Options

This is done automatically through the `__C51__` symbol defined by the Keil Cx51 C compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 57. It is similar to that used by Keil for the standard Cx51 libraries.

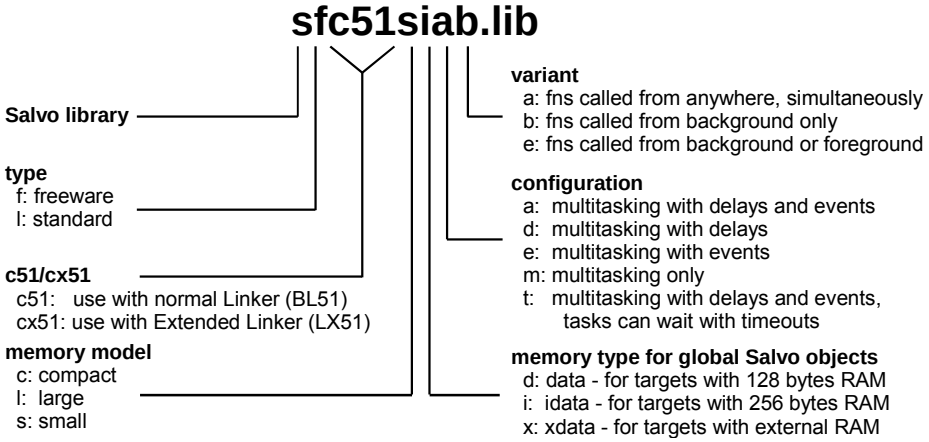


Figure 57: Salvo Library Nomenclature – Keil Cx51 C Compiler

When using the BL51 standard linker, the appropriate libraries contain `c51` in their names. When using the LX51 extended linker, the appropriate libraries contain `cx51` in their names. A mismatch will cause DATA TYPES DIFFERENT warnings.

128 Cx51 referes to both the C51 and Cx51 compilers.

Example salvocfg.h

The library type, globals, configuration and variant must be specified when using a Salvo library for the Keil Cx51 C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 57 is shown in Listing 52.

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_GLOBALS       OSI
#define OSLIBRARY_CONFIG        OSA
#define OSLIBRARY_VARIANT       OSB
```

Listing 52: Example salvocfg.h for Library Builds – Keil Cx51 C Compiler

It is not necessary to indicate which memory model the library was built for (a mismatch will generate a link-time error with an obvious message). Failing to define `OSLIBRARY_GLOBALS` will result in linker errors 103 and 118.

Build Settings

The Salvo libraries for the Keil Cx51 C compiler are compiled with the configuration options listed in Table 31. All other configuration options that are not affected by the library's *configuration* (see below) are at their default values.

configuration	value
Event flags:[129]	1
Events:	5
Message queues:[130]	1
Tasks:	3
Memory model:	see Table 32
Salvo objects:	see Table 33
System tick counter:	available, 32 bits
Watchdog timer:	not cleared
Task priorities:	see Table 14
Idling function:	see Table 14

Table 31: Configuration Options Used to Build Salvo Libraries – Keil Cx51 C Compiler

129 Each event flag has RAM allocated to its own event flag control block.

130 Each message queue has RAM allocated to its own message queue control block.

Various memory models can be used when building applications with the Keil Cx51 C compiler. You must use the Salvo library that matches the memory model of the rest of your application. The memory model codes are listed in Table 32.

`c:`	*C*ompact memory model.
`l:`	*L*arge memory model.
`s:`	*S*mall memory model (C51 default).

Table 32: Memory Model Codes for Salvo Libraries – Keil Cx51 C Compiler

You can choose the memory type for the Salvo objects in your application by choosing the appropriate library. `data` type objects can be accessed the fastest, but consume precious RAM on 8051s with only 128 bytes of RAM. `idata` type objects can be used on 8051s with 256 bytes of RAM. `xdata` type objects will be placed in external RAM, which will result in slower accesses. The memory type codes are listed in Table 33.

`d / OSD:`	Salvo objects are declared as type `data`, and will be located in the first 128 bytes of internal RAM.
`I / OSI:`	Salvo objects are declared as type `idata`, and will be located in the first 256 bytes of internal RAM.
`x / OSX:`	Salvo objects are declared as type `xdata`, and will be located in external RAM.

Table 33: Salvo Object Memory Type Codes for Salvo Libraries – Keil Cx51 C Compiler

Notes

There are 540 Salvo libraries for the Keil Cx51 C compiler. 270 are for use with the BL51 linker, and 270 for use with the LX51 linker.

The *b*- and *e*-variant libraries have `OSPRESERVE_INTERRUPT_MASK` set to `FALSE` and `TRUE`, respectively. The *a*-variant libraries also have `OSPRESERVE_INTERRUPT_MASK` set to `TRUE` and use reentrant functions to allow simultaneous calling from two or more processes.

Motorola M68HC11: ImageCraft ICC11 C Compiler

The ImageCraft ICC11 C compiler is a Win32-based cross-compiler for use with the Motorola M68HC11 microcontroller. This compiler is commonly used within the ImageCraft ICC IDE, with a Motorola 68HC11 EVB or similar development system connected via the IDE's terminal interface.

Compiler and Target Processor Configuration Options

This is done automatically through the `__IMAGECRAFT__` and `_HC11` symbols defined by the ImageCraft ICC11 C compiler. However, the IDE does not define these symbols when deriving information for its built-in `make` utility. Therefore you *must* specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file when using the ICC11 IDE.

Nomenclature

The libraries follow the naming convention shown in Figure 58.

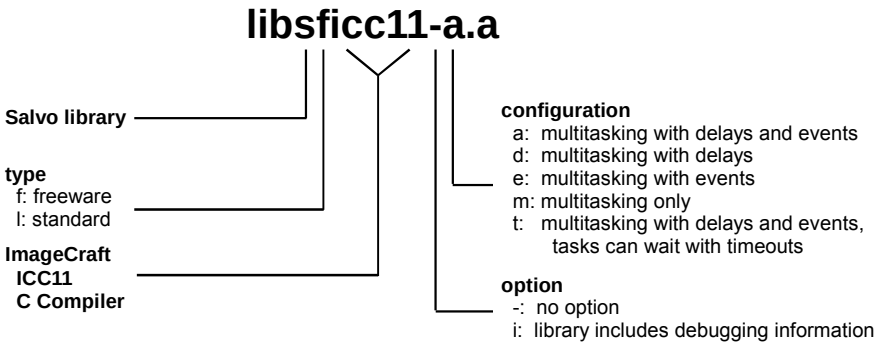


Figure 58: Salvo Library Nomenclature – ImageCraft ICC11 C Compiler

Example salvocfg.h

The library type and configuration must be specified when using a Salvo library for the ImageCraft ICC11 C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 58 is shown in Listing 54.

```
#define OSCOMPILER              OSIMAGECRAFT
#define OSTARGET                OSM68HC11

#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_CONFIG        OSA
```

Listing 53 Example salvocfg.h for Library Builds – ImageCraft ICC11 C Compiler

Build Settings

The Salvo libraries for the ImageCraft ICC11 C compiler are compiled with the configuration options listed in Table 34. All other configuration options by the library's configuration (see below) are at their default values.

configuration	value
Tasks:	3
Events:	5
Event flags:	1
Message queues:	1
Delay sizes:	see Table 14
Idling function:	see Table 14
Task priorities:	see Table 14
Watchdog timer:	cleared in `OSSched()`
System tick counter:	available, 32 bits

Table 34: Configuration Options Used to Build Salvo Libraries – ImageCraft ICC11 C compiler

Salvo Pro users can select between two sets of libraries – standard libraries, and standard libraries incorporating source-level debugging information.[131] The latter have been built with the ImageCraft ICC11 C compiler's `+g` command-line options. This adds source-level debugging information to the libraries, making them ideal for source-level debugging and stepping in the ICC11 debugger. To use these libraries, simply select one that includes the debugging information (e.g. `libslicc11it.a`) instead of one without (e.g. `libslicc11-t.a`) in your ICC11 project.

131 The Salvo libraries provided with Salvo Lite and LE for M68HC11 do not contain ICC11-debugger-compatible debugging information because this requires the inclusion of source file listings.

Processors

The M68HC11's core architecture is consistent throughout the various available variants. Therefore a single set of Salvo libraries for the ImageCraft ICC11 C compiler serves all M68HC11 variants.

Notes

There are 15 Salvo libraries for the ImageCraft ICC11 C compiler.

The ImageCraft ICC11 C compiler expects libraries to be in `\icc\lib`. Therefore the Salvo installer places its libraries for ICC11 in both `\salvo\lib` and `\icc\lib`.

TI's MSP430: Archelon / Quadravox AQ430 C Compiler

The Archelon AQ430 C compiler is a Win32-based cross-compiler for use with 16-bit MSP430 ultra-low-power microcontrollers from Texas Instruments. This compiler is commonly used within the Quadravox qvarch430tools IDE. A directly-connected MSP430 Flash Emulation Tool (FET) is required for debugging from within the IDE. The AQ430 toolset is a joint development between Archelon and Quadravox.

Compiler and Target Processor Configuration Options

This is done automatically through the `__ARCHELON__` and `__AQCOMPILER__` symbols defined by the Archelon AQ430 C compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 59.

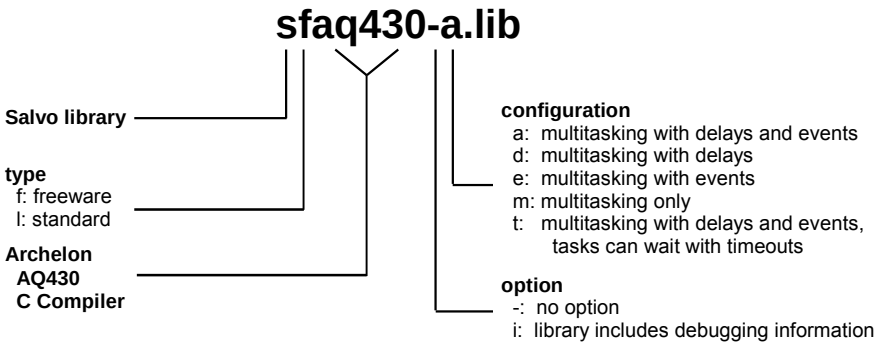


Figure 59: Salvo Library Nomenclature – Archelon AQ430 C Compiler

Example salvocfg.h

The library type and configuration must be specified when using a Salvo library for the Archelon AQ430 C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 59 is shown in Listing 54.

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_CONFIG        OSA
```

Listing 54 Example salvocfg.h for Library Builds – Archelon AQ430 C Compiler

Build Settings

The Salvo libraries for the Archelon AQ430 C compiler are compiled with the configuration options listed in Table 35. All other configuration options by the library's configuration (see below) are at their default values.

configuration	value
Tasks:	3
Events:	5
Event flags:	1
Message queues:	1
Delay sizes:	see Table 14
Idling function:	see Table 14
Task priorities:	see Table 14
Watchdog timer:	cleared in `OSSched()`
System tick counter:	available, 32 bits

Table 35: Configuration Options Used to Build Salvo Libraries – Archelon AQ430 C compiler

Salvo Pro users can select between two sets of libraries – standard libraries, and standard libraries incorporating source-level debugging information.[132] The latter have been built with the Archelon AQ430 C compiler's `+dbg` and `asm=+coff` command-line options. This adds source-level debugging information to the libraries, making them ideal for source-level debugging and stepping in the AQ430 debugger. To use these libraries, simply select one that includes the debugging information (e.g. `slaq430it.lib`) instead of one without (e.g. `slaq430-t.lib`) in your AQ430 project and Options → Project → Miscellaneous → Modify File Names in Object Files.

[132] The Salvo libraries provided with Salvo Lite and LE for MSP430 do not contain AQ430-debugger-compatible debugging information because this requires the inclusion of source file listings.

Processors

The MSP430's core architecture is consistent throughout the entire family. Therefore a single set of Salvo libraries for the Archelon AQ430 C compiler serves all MSP430 variants.

Notes

There are 15 Salvo libraries for the Archelon AQ430 C compiler.

TI's MSP430: IAR MSP430 C Compiler

The IAR MSP430 C compiler is a Win32- and UNIX-based cross-compiler for use with 16-bit MSP430 ultra-low-power microcontrollers from Texas Instruments. This compiler is commonly used within IAR's Embedded Workbench IDE and C-SPY debugger.

Compiler and Target Processor Configuration Options

This is done automatically through the `__IAR_SYSTEMS_ICC` and `__TID__` symbols defined by the IAR MSP430 C compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 60.

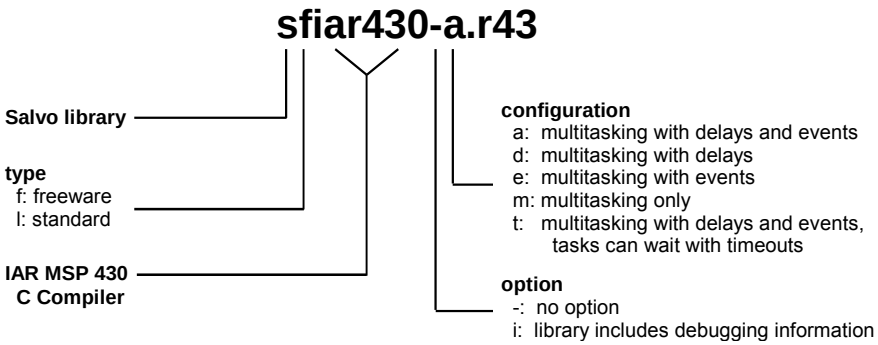


Figure 60: Salvo Library Nomenclature – IAR MSP430 C Compiler

Example salvocfg.h

The library type and configuration must be specified when using a Salvo library for the IAR MSP430 C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 60 is shown in Listing 55.

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_CONFIG        OSA
```

Listing 55 Example salvocfg.h for Library Builds – IAR MSP430 C Compiler

Build Settings

The Salvo libraries for the IAR MSP430 C compiler are compiled with the configuration options listed in Table 36. All other configuration options by the library's configuration (see below) are at their default values.

configuration	value
Tasks:	3
Events:	5
Event flags:	1
Message queues:	1
Delay sizes:	see Table 14
Idling function:	see Table 14
Task priorities:	see Table 14
Watchdog timer:	cleared in `OSSched()`
Salvo objects:	in `DATA` memory[133]
System tick counter:	available, 32 bits

Table 36: Configuration Options Used to Build Salvo Libraries – IAR MSP430 C compiler

Salvo Pro users can select between two sets of libraries – standard libraries, and standard libraries incorporating source-level debugging information.[134] The latter have been built with the IAR MSP430 C compiler's `-re` command-line option. This adds source-level debugging information to the libraries, making them ideal for source-level debugging and stepping in C-SPY. To use these libraries, simply select one that includes the debugging information (e.g. `sliar430it.r43`) instead of one without (e.g. `sliar430-t.r43`) in your Embedded Workbench project.

Processors

The MSP430's core architecture is consistent throughout the entire family. Therefore a single set of Salvo libraries for the IAR MSP430 C compiler serves all MSP430 variants.

---

133 Salvo objects can be placed in non-volatile RAM by setting `OSLOC_ALL` to `no_init` and defining the `NO_INIT` segment in the XLINK command file.

134 The Salvo libraries provided with Salvo Lite and LE for MSP430 do not contain C-SPY-compatible debugging information because this requires the inclusion of source file listings.

Notes

There are 15 Salvo libraries for the IAR MSP430 C compiler.

TI's MSP430: ImageCraft ICC430 C Compiler

The ImageCraft ICC430 C compiler is a Win32-based cross-compiler for use with 16-bit MSP430 ultra-low-power microcontrollers from Texas Instruments. A directly-connected MSP430 Flash Emulation Tool (FET) is required for debugging from within the IDE.

Compiler and Target Processor Configuration Options

This is done automatically through the `__IMAGECRAFT__` and `_MSP430` symbols defined by the ImageCraft ICC430 C compiler. However, the IDE does not define these symbols when deriving information for its built-in `make` utility. Therefore you *must* specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file when using the ICC430 IDE.

Nomenclature

The libraries follow the naming convention shown in Figure 61.

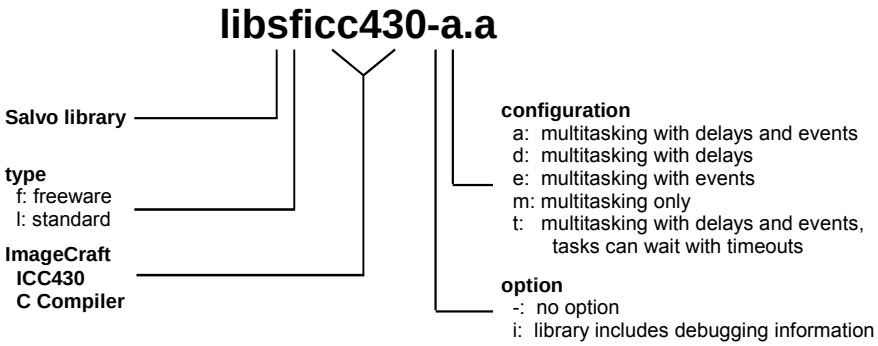


Figure 61: Salvo Library Nomenclature – ImageCraft ICC430 C Compiler

Example salvocfg.h

The library type and configuration must be specified when using a Salvo library for the ImageCraft ICC430 C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 61 is shown in Listing 56.

```
#define OSCOMPILER              OSIMAGECRAFT
#define OSTARGET                OSMSP430

#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_CONFIG        OSA
```

Listing 56 Example salvocfg.h for Library Builds – ImageCraft ICC430 C Compiler

Build Settings

The Salvo libraries for the ImageCraft ICC430 C compiler are compiled with the configuration options listed in Table 37. All other configuration options by the library's configuration (see below) are at their default values.

configuration	value
Tasks:	3
Events:	5
Event flags:	1
Message queues:	1
Delay sizes:	see Table 14
Idling function:	see Table 14
Task priorities:	see Table 14
Watchdog timer:	cleared in `OSSched()`
System tick counter:	available, 32 bits

Table 37: Configuration Options Used to Build Salvo Libraries – ImageCraft ICC430 C compiler

Salvo Pro users can select between two sets of libraries – standard libraries, and standard libraries incorporating source-level debugging information.[135] The latter have been built with the ImageCraft ICC430 C compiler's `+g` command-line options. This adds source-level debugging information to the libraries, making them ideal for source-level debugging and stepping in the ICC430 debugger. To use these libraries, simply select one that includes the debugging information (e.g. `libslicc430it.a`) instead of one without (e.g. `libslicc430-t.a`) in your ICC430 project.

[135] The Salvo libraries provided with Salvo Lite and LE for TI's MSP430 do not contain ICC430-debugger-compatible debugging information because this requires the inclusion of source file listings.

Processors

The MSP430's core architecture is consistent throughout the various available variants. Therefore a single set of Salvo libraries for the ImageCraft ICC430 C compiler serves all MSP430 variants.

Notes

There are 15 Salvo libraries for the ImageCraft ICC430 C compiler.

The ImageCraft ICC430 C compiler expects libraries to be in `\icc\lib`. Therefore the Salvo installer places its libraries for ICC430 in both `\salvo\lib` and `\icc\lib`.

ARC / VAutomation V8-µRISC: HI-TECH V8C C Compiler

Compiler and Target Processor Configuration Options

This is done automatically through the `HI_TECH_C` and `_V8` symbols defined by the HI-TECH V8C C compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 62. The types, configurations and variants are identical to those of the PICC libraries and are described above in *Microchip PICmicro MCUs: HI-TECH PICC Compiler* .

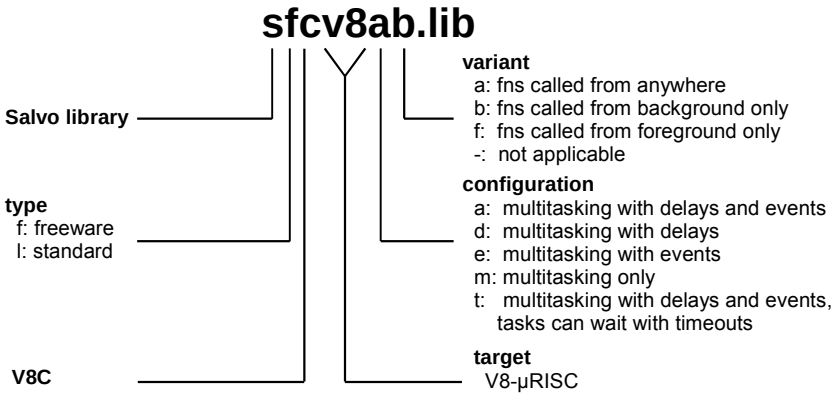


Figure 62: Salvo Library Nomenclature – HI-TECH V8C C Compiler

Example salvocfg.h

The library type, configuration and variant must be specified when using a Salvo library for the HI-TECH V8C C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 62 is shown in Listing 57.

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_CONFIG        OSA
```

```
#define OSLIBRARY_VARIANT        OSB
```

Listing 57: Example salvocfg.h for Library Builds – HI-TECH V8C C Compiler

Build Settings

The Salvo libraries for the HI-TECH V8C C compiler are compiled with the configuration options listed in Table 38. All other configuration options that are not affected by the library's *configuration* (see below) are at their default values.

configuration	value
Event flags:[136]	1
Events:	5
Message queues:[137]	1
Tasks:	3
Salvo objects:[138]	`persistent`
System tick counter:	available, 32 bits
Task priorities:	see Table 14
Idling function:	see Table 14
Interrupt level:[139]	0

Table 38: Configuration Options Used to Build Salvo Libraries – HI-TECH V8C C Compiler

Note `OSInit()` *must* be used in all applications that use these libraries. Without it, Salvo's variables will be uninitialized, with unpredictable results.

Notes

There are 30 Salvo libraries for the HI-TECH V8C C compiler.

Usage of the freeware and standard libraries for HI-TECH V8C C compiler is substantially similar to that of the libraries for the HI-TECH family of PICC C compilers (see above). Please refer to

136 Each event flag has RAM allocated to its own event flag control block.

137 Each message queue has RAM allocated to its own message queue control block.

138 By making Salvo's variables `persistent`, the V8C compiler is able to omit some initialization code and thus reduce ROM requirements.

139 Argument for V8C"s `#pragma interrupt_level` for those services that can be called from within an ISR.

PICC-specific Application Notes and other Salvo documentation for related information.

x86 PCs: GNU C Compiler (gcc)

Compiler and Target Processor Configuration Options

This is done automatically through the `__GNUC__` symbol defined by the GNU C compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 63.

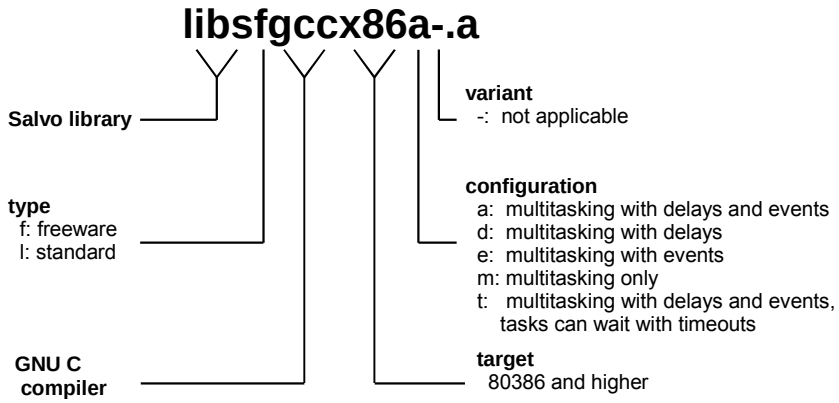


Figure 63: Salvo Library Nomenclature – GNU C Compiler

Example salvocfg.h

The library type, configuration and variant must be specified when using a Salvo library for the GNU C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 63 is shown in Listing 58.

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_CONFIG        OSA
#define OSLIBRARY_VARIANT       OSNONE
```

Listing 58: Example salvocfg.h Library Builds – GNU C Compiler

Build Configuration

The Salvo libraries for the GNU C compiler are compiled with the configuration options listed in Table 39.

configuration	value
Event flags:[140]	25
Events:	25
Message queues:[141]	25
Tasks:	25
System tick counter:	available, 32 bits
Task priorities:	enabled
Idling function:	enabled
Delay size:	8 bits
Timeouts:	enabled

Table 39: Configuration Options Used to Build Salvo Libraries – GNU C Compiler

Notes

There are 10 Salvo libraries for the GNU C compiler.

These libraries are compiled for 80386 and later target processors.

These libraries are simple static libraries. They can be linked to your Salvo application like this:

```
gcc main.c -I../../include -I./ -o main.exe
  -L../../library/ -lsfgccx86t- -mno-cygwin
```

Listing 59: Linking a Salvo Application to a Salvo Library for the GNU C Compiler

GNU Public License (GPL) Issues

Applications that include Salvo code *may not* be distributed under the terms of the GPL – see the Pumpkin Salvo Software License Agreement and the GNU Public License at http://www.gnu.org/ for more information.

140 Each event flag has RAM allocated to its own event flag control block.

141 Each message queue has RAM allocated to its own message queue control block.

Therefore if you create a Salvo application using gcc,[142] and it contains any component bound by the GPL (e.g. the Cygwin Win32 DLL), you may not distribute it in any form.

[142] Or any other compiler, but gcc is often in close proximity to other GPL'd software.

x86 PCs: Metrowerks CodeWarrior C Compiler

Note All of Salvo's CodeWarrior libraries are freeware libraries – there are no standard libraries. Unlike other Salvo freeware libraries, they are compiled *without* any limit placed on Salvo objects, etc. In other words, they are "free" standard libraries, and they are included in all Salvo distributions.

Compiler and Target Processor Configuration Options

This is done automatically through the `__MWERKS__` symbol defined by the Metrowerks CodeWarrior C compiler. Therefore there's no need to specify `OSCOMPILER` and `OSTARGET` in your `salvocfg.h` file.

Nomenclature

The libraries follow the naming convention shown in Figure 64. The nomenclature is explained in detail below.

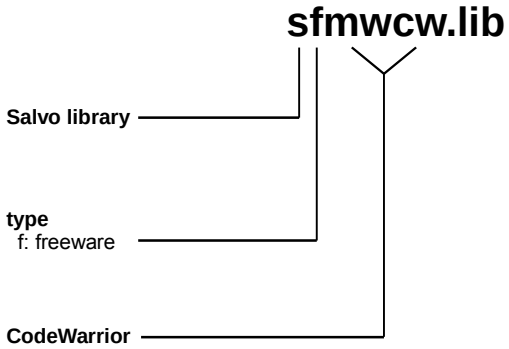


Figure 64: Salvo Library Nomenclature – Metrowerks CodeWarrior C Compiler

Example salvocfg.h

The library type must be specified when using a Salvo library for the Metrowerks CodeWarrior C compiler. The appropriate `salvocfg.h` for the library illustrated in Figure 64 is shown in Listing 60:

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
```

```
#define OSLIBRARY_CONFIG          OSNONE
#define OSLIBRARY_VARIANT         OSNONE
```

Listing 60: Example salvocfg.h for use with Salvo Libraries – Metrowerks CodeWarrior C Compiler

Build Configuration

The Salvo library for the Metrowerks CodeWarrior C compiler is compiled with the configuration options listed in Table 40.

configuration	value
Event flags:[143]	25
Events:	25
Message queues:[144]	25
Tasks:	25
System tick counter:	available, 32 bits
Task priorities:	enabled
Idling function:	enabled
Delay size:	8 bits
Timeouts:	enabled

Table 40: Configuration Options Used to Build Salvo Library – Metrowerks CodeWarrior C Compiler

Notes

There is only 1 Salvo library for the Metrowerks CodeWarrior C compiler.

Unlike other Salvo freeware libraries, the freeware libraries for the Metrowerks CodeWarrior C compiler are compiled with `OSSET_LIMITS` set to `FALSE`, and therefore the number of Salvo objects they support is unlimited. If you wish to use more than the library's default number of objects, see *Overriding Default RAM Settings*, above.

This library is compiled for "Generic 80x86" target processors.

143 Each event flag has RAM allocated to its own event flag control block.

144 Each message queue has RAM allocated to its own message queue control block.

Rebuilding the Libraries

The Salvo libraries are generated with GNU make in the bash shell.[145] If you have Salvo Pro or Salvo Developer you can rebuild the libraries using the makefiles in the `salvo/src` directory.

Note No provision for external paths to compilers, etc. is provided in the makefiles. If you have installed your compiler(s) in places that differ from those specified in the makefiles, you will need to edit the makefile(s) for a successful compile.

Note The makefiles are designed to run from the `salvo/src` directory.

Customizing the Libraries

You can rebuild the Salvo libraries to a configuration that differs from the standard build.[146] Probably the easiest way to do this is to modify `salvo/inc/salvolib.h` by changing and/or inserting configuration options in places appropriate for your compiler and/or target.

For example, to rebuild the Salvo libraries for the HI-TECH PICC compiler and mid-range PICmicro® MCU targets with Salvo's global variables placed in RAM bank 3 instead of RAM bank 1, you would change:

```
/*                                                  */
/* Begin PIC16 library special settings and
   overrides                                        */
/*                                                  */
#elif ( OSTARGET == OSPIC16 )

#define OSCLEAR_WATCHDOG_TIMER()  asm(" clrwdt")
#if (_BANKBITS_ == 0)
#define OSLOC_ALL                 persistent
#else
#define OSLOC_ALL                 bank1 persistent
#endif
```

in `salvolib.h` to

[145] Bourne-again shell, a Unix command language interpreter.

[146] Note that Pumpkin cannot provide support for libraries that differ from those provided in the Salvo distributions.

```
/*                                                  */
/* Begin PIC16 library special settings and
   overrides                                        */
/*                                                  */
#elif ( OSTARGET == OSPIC16 )

#define OSCLEAR_WATCHDOG_TIMER()  asm(" clrwdt")
#if (_BANKBITS_ == 0)
#define OSLOC_ALL                 persistent
#else
#define OSLOC_ALL                 bank3 persistent
#endif
```

and rebuild the libraries.

Linux/Unix Environment

To rebuild one or more Salvo libraries in the bash shell, simply issue the commands:

```
$: cd /salvo/src
$: make –f Makefile target
```

Listing 61: Rebuilding Salvo Libraries in a Linux/Unix Environment

where target is either a specific library (e.g. `sfp42Cab.lib`) or a family of libraries (e.g. `p42C`). Naturally, you will need the compiler associated with the Salvo libraries you're (re-)building.

A list of target groups can be obtained by issuing the commands:

```
$: cd /salvo/src
$: make –f Makefile liblist
```

Listing 62: Obtaining a List of Library Targets in the Makefile

Win32 Environment

To rebuild Salvo libraries in a Win32 environment, you will need a bash shell along with GNU make. One free source for both is the Cygwin bash shell. Another is the MinGW project, along with associated utilities. You can then use the method shown in Listing 61, above.

Note Due to a problem with recursive make in the Cygwin bash shell, attempts to make large numbers of Salvo libraries are likely

to fail when using `Makefile`. Therefore, Salvo includes some bash scripts that avoid calling make recursively. See *salvo\src\makeXyz.bat* below for more information.

Apparently, this problem does not exist with MinGW.

Note The Windows 95 / 98 environment is *not recommended* for generating Salvo libraries via the Cygwin bash shell. Consider migrating to Windows NT / 2000 / XP if you plan on building Salvo libraries.

Makefile Descriptions

salvo\src\Makefile

This makefile uses a regular expression to parse the name of the desired library or libraries. It then calls make recursively using `Makefile2` to generate one or more libraries.

salvo\src\Makefile2

This makefile actually invokes the appropriate compiler, with defined symbols corresponding to the type, target, configuration and variant desired.

salvo\src\targets.mk

This include file contains the names of all valid Salvo libraries, grouped by target processor.

salvo\src\makeXyz.bat

These MS-DOS batch files can be called from *within the bash shell* to build entire groups of Salvo libraries. By using only `Makefile2` and not calling make recursively, they can be used to build all of Salvo's libraries within the Cygwin environment.

For example, to build all of the PIC16 PICmicro libraries for use with the HI-TECH PICC compiler, you would issue the commands:

```
$: cd /salvo/src
$: makep4xx.bat
```

Listing 63: Building the Salvo PICC Libraries for mid-range PICmicros in the Win32 Environment

Chapter 9 • Performance

Introduction

In this chapter we'll present examples of Salvo's performance using actual demonstration and test programs. Use this chapter get an accurate idea of how much memory your application will require and how fast it will perform when multitasking with Salvo.

Measuring Performance

You probably want to know how much memory (ROM and RAM) Salvo uses, and how quickly Salvo performs actions like context switching. In other words, you're interested in Salvo's compile-time performance as well as its run-time performance.

Because Salvo is so highly configurable, it's impossible to present a single set of universally applicable performance figures. Instead, we'll present figures using a couple of representative test systems. If your particular application doesn't match any of the test systems, you can always do your own testing using the test programs as a guide.

Where hard numbers are presented, all details concerning Salvo configuration, the compiler in use, the target processor, and any other pertinent details will be given. All test programs are provided in the standard Salvo installation, in both source code (with comments) and object code formats. They can be found in the `salvo\test` directory.

Performance Examples

Before we begin an in-depth look at Salvo's compile-time and run-time performance, here are the results of some simple test programs.

Test Systems

Three different test systems will be used to illustrate Salvo's real-world performance in this chapter. Systems A and B are based on mid-range and high-end 8-bit processors, respectively, while system C is representative of a 450MHz Pentium-II-class PC. The systems are summarized below:

Feature	A	B	C
Processor	Microchip PIC16C77	Microchip PIC17C756	AMD K6-2/450
Architecture	Harvard RISC	Harvard RISC	RISC86 Superscalar
Clock speed	4MHz	16MHz	450MHz
Instruction cycle time[147]	1us	250ns	varies
RAM	368 bytes	902 bytes	128MB installed
ROM	8K 14-bit words	64K[148] 16-bit words	n/a
Call...return stack depth	8 levels	16 levels	unlimited
Testing environment	Microchip PICDEM-2 demo board	proprietary data acquisition system	PC100-class motherboard
Compiler Used	HI-TECH PICC	HI-TECH PICC	Mix Power C
Pointer size[149]	1 byte	1 byte	2 bytes

Table 41: Test System Overview

Note Performance figures are not unique to the test system. For example, Salvo's performance in system A (B) is representative of its performance in much of Microchip's PIC16 (PIC17) family.

Note Individual performance measures may not be available for all systems.

Pumpkin uses additional test systems for various purposes, including code testing and verification, porting to new targets and com-

147 All of the PIC16C6X and PIC17C7XX instructions are single-cycle instructions except for program branches, which require two cycles.

148 External program memory in microprocessor mode. Has 16K words of on-chip program memory in microcontroller mode.

149 PIC16C6X and PIC17C75X RAM is banked.

pilers, and example code. Further information on test systems can be found in Appendix C.

Test Configurations

In this chapter, different compile-time configurations will be used when characterizing Salvo's performance. Generally speaking, as you add more functionality to your Salvo application (e.g. by adding support for delays), the ROM and RAM requirements will increase, and the time required for certain user services may vary.

The test configurations used in this chapter are shown below.

Configuration	Multitasking	Delays	Events	Time-outs
I	√			
II	√	√		
III	√		√	
IV	√	√	√	
V	√	√	√	√

Table 42: Features Enabled in Test Configurations I-V

Test Programs

Tests 1 through 5 show the ROM and RAM requirements for some sample multitasking applications using Salvo. These are real, working programs with a structure that's representative of typical applications, complete with calls to Salvo services for every task and event. They do not contain any user code – i.e. `main()` and the tasks call only Salvo services.

Test 1 explicitly creates and starts 8 tasks that do nothing but context-switch. Test 2 adds the system timer with support for 8-bit delays, and has the tasks delaying instead of simply running. Test 3 adds to test 1 by supporting events and having six tasks wait for semaphores to be signaled. Test 4 is a combination of tests 2 and 3. Test 5 has tasks waiting with timeouts. Tests 1 and 3 do not use interrupts.

Test programs 1 through 5 are compiled with test configurations I through V, respectively.

Note The memory requirements shown in these tests are the total memory required by the test programs, not just by Salvo. Startup code, variable initialization and other runtime modules, as well as temporary variables, function parameters, auto variables, interrupt handlers, etc. are placed in ROM and RAM by the compiler and are included in the totals below. The actual size of Salvo's own functions and variables are shown in ().

Test Program	A	B
1	536 (318) words 45 (34) bytes	560 words 45 bytes
2	781 words 72 (43) bytes	
3	965 (555) words 72 (53) bytes	
4	1209 words 96 (XX) bytes	
5	1463 words 111 (XX) bytes	

Table 43: ROM and RAM Usage for Test Programs 1-5 in Test Systems A & B

Tests 6 through 10 give the context-switch time and rates for some example configurations. While an RTOS' context-switch rate should not be viewed as a particularly good indication of overall performance, it can provide some insight towards the performance of your multitasking application. Test 6 runs an idle task over and over again. Test 7 runs five tasks of equal priority, i.e. they round-robin. Test 8 runs five tasks of unique priorities, i.e. that the highest-priority task is always running, and the rest remain eligible to run. All three tests count the number of context switches. The observed context switching rates and times are shown below:

Test Program	A	B	C
6	5,682/sec 176us	28,169/sec 35.50us	1,120,000/sec (approx.)
7	3,846/sec 260us	17,937/sec 55.75us	XXX
8	5,435/sec 184us	26,846/sec 37.25us	XXX

Table 44: Context-Switching Rates & Times for Test Programs 6-10 in Test Systems A-C

Compile-time Performance

Salvo's compile-time design goals are, in order:

- to be as portable as possible,
- to minimize RAM usage,
- to minimize code size (from C),
- to minimize call ... return stack depth,
- to provide for flexibility in supporting different features,
- to offer optional optimizations to improve execution speed and other performance issues, and
- minimize register usage.

As a programmer, you choose the configuration options and Salvo does the rest.

Code Size (ROM)

The size of the Salvo code in your application depends primarily on:

- how Salvo is configured,
- the efficiency of your compiler and target processor's instruction set,
- how you compile your application, and
- which Salvo services you use in your application.

Usually Salvo code will reside in ROM. Some configuration options, e.g. OS`ENABLE_TIMEOUTS`, can have a substantial impact on the size of the Salvo code. This usually occurs when you trade off one feature (e.g. larger code size) against another (improved speed) by configuring Salvo accordingly. Some other configuration op-

tions, e.g. `OSENABLE_STATISTICS`, cause the Salvo code to grow in size simply because additional features have been enabled. Generally speaking, the default configuration file `salvocfg.h` will result in the smallest libraries being generated when you compile the Salvo source code.

Compiler and instruction set efficiency can have a major impact on the size of the Salvo code. Some compilers are better than others at creating tight and efficient code, and some processors achieve higher code densities than others. While you can't do much about your processor's instruction set, you can take advantage of various compiler offerings. It is recommended that you initially compile the Salvo code with your compiler's optimizations turned off. Once you've verified that the code being generated is correct and works properly in your application, you're encouraged to use your compiler's optimization features to shrink the size of the Salvo code. In some instances you may see 15-20% reductions in the size of the code.

How you compile and link the Salvo code may affect its size in your application. Unless you extract the Salvo functions you use by linking to a precompiled library, you may end up with Salvo functions in your code that you never use. In a large application this may not be an issue, but in a small application this may make the difference between being able to fit your application in ROM, and running out of code space. See *Chapter 4 • Tutorial* for more details on how to compile and link Salvo to your application most efficiently.

Every Salvo applications requires at least a few Salvo functions, e.g. `OSInit()` and `OSSched()`. Some of the multitasking services, like `OSCreateTask()`, will be in every application. Others, e.g. `OSSetPrio()`, may or may not be present. As you add features like delays, events and support for timeouts to your application, the size of the Salvo code will grow. Generally speaking, the Salvo code will grow as you enable and/or use the following features, in the order shown below.

- multitasking only
- multitasking with delays
- multitasking with events
- multitasking with delays and events
- multitasking with delays and events, with timeouts supported

Ultimately, the simplest way to obtain the code size of your Salvo application is to compile it and look at the compiler's output. By changing configuration options and your compiler's optimization levels you can minimize the code size to fit in your application.

Tip If your target processor has limited memory, you may find it useful to initially compile your code for a similar processor with more memory. Then, by judicious choice of configuration options and compiler optimizations, you can try to squeeze your application down to a smaller size.

Note Code size will not change as you change the number of tasks and events (if defined) in `salvocfg.h`.

Variables (RAM)

All Salvo RAM is allocated at compile time, and only the RAM that's need for the configuration you've chosen will be allocated. Salvo makes extremely efficient use of RAM, without storing redundant or unnecessary information. A minimal multitasking Salvo application using the default `salvocfg.h` will have the following RAM requirements:

- 2 RAM pointers,
- 1 byte, a task (ROM) pointer and 1 queue (RAM) pointer for each task.

Generally speaking, Salvo's RAM requirements will increase as you enable and/or use the following features, in the order shown below:

- multitasking only,
- multitasking with delays,
- multitasking with events,
- multitasking with delays and events, and
- multitasking with delays and events, with timeouts supported.

Depending on the configuration you've chosen, Salvo will require up to a maximum of

- 3 RAM pointers,
- a task (ROM) pointer, 1-4 queue (RAM) pointers and 1-6 bytes for each task,

- 1 queue (RAM) pointer and 1-3 bytes[150] for each event, and
- memory for options (e.g. context switch counter)

in RAM. For example, enabling 32-bit delays will add 4 bytes of RAM per task. Enabling the idle function hook and the collecting of statistics with 16-bit counters will add 8 bytes to Salvo's memory requirements. Consult *Chapter 5 • Configuration* for more details on the memory requirements of Salvo's configuration options.

Note The amount of RAM required per task is independent of the number of events, and similarly the amount of RAM required per event is independent of the number of tasks.

Below are the Salvo memory requirements (in bytes of RAM) for a complete multitasking application with up to 8 tasks, 8-bit delays, 6 events and support for timeouts:

Configuration	A	B	C
I	35	35	XXX
II	43	43	XXX
III	53	53	XXX
IV	61	61	
V	77	77	XXX

Table 45: RAM Requirements for Configurations I-V in Test Systems A-C

Note Salvo supports the placement of RAM variables in separate banks for processors with banked RAM.

The only configuration options that may affect RAM requirements without affecting code size are those which specify the size of a particular numeric field, e.g. `OSBYTES_OF_DELAYS`.

Run-time Performance

Salvo's primary design goal is for maximum run-time performance while respecting the compile-time design goals. Salvo was written first and foremost to fit in small, inexpensive processors where memory is at a premium.

150 Assumes 8- or 16-bit message pointers.

Representative cycle counts for the entire run-time performance section are shown for test system A, with each instruction cycle lasting 1us (except program branches, which take 2us).

Other processors and compilers will generate different results. However, since the underlying algorithms are independent of the processor and compiler, the cycle counts for other systems are likely to have similar overall behavior, but will be scaled according to the operating speed of the processor.

Salvo incorporates support for some speed optimizations (e.g. `OSSPEEDUP_QUEUEING`) that were felt to be worth the added memory requirements. As a programmer you can choose to use or not use these optimizations by selecting the appropriate configuration option(s).

Note Run-time performance figures are solely for Salvo code and do not include the effects of your application, e.g. non-Salvo interrupts which may occur while a Salvo service is executing.

Tip The execution times for system A represent both the number of instruction cycles, and the execution time in microseconds.

Speeds of User Services

The execution times required to perform Salvo user services are a combination of fixed times and variable queueing operations times, where applicable. They are shown below, in instruction cycles, along with the time that interrupts are disabled.

Note The execution times below are for the default Salvo configuration unless otherwise noted. Execution times include the time to pass parameters to the service.

OS_Delay()

	duration	interrupts disabled
min	65 + t_InsDelayQ	56 + t_InsDelayQ
max	65 + t_InsDelayQ	56 + t_InsDelayQ

Table 46: OS_Delay() Execution Times

OS_Destroy()

	duration	interrupts disabled
min	38	0
max	38	0

Table 47: OS_Destroy() Execution Times

OS_Prio()

	duration	interrupts disabled
min	65	0
max	65	0

Table 48: OS_Prio() Execution Times

OS_Stop()

	duration	interrupts disabled
min	43	0
max	43	0

Table 49: OS_Stop() Execution Times

OS_WaitBinSem()

	duration	interrupts disabled
min	?	?
max	? + t_InsPrioQ	? + t_InsPrioQ

	condition
min	Binary semaphore is 1.
max	Binary semaphore is 0.

Table 50: OS_WaitBinSem() Execution Times

OS_WaitMsg()

	duration	interrupts disabled
min	85	61
max	128 + t_InsPrioQ	85 + t_InsPrioQ

	condition
min	Message is available.
max	Message is not available.

Table 51: OS_WaitMsg() Execution Times

OS_WaitMsgQ()

	duration	interrupts disabled
min	?	?
max	? + t_InsPrioQ	? + t_InsPrioQ

	condition
min	There is at least one message in the message queue.
max	The message queue is empty.

Table 52: OS_WaitMsgQ() Execution Times

OS_WaitSem()

	duration	interrupts disabled
min	71	54
max	114 + t_InsPrioQ	88 + t_InsPrioQ

	condition
min	Semaphore is non-zero.
max	Semaphore is 0.

Table 53: OS_WaitSem() Execution Times

OS_Yield()

	duration	interrupts disabled
min	22	0
max	22	0

Table 54: OS_Yield() Execution Times

OSCreateBinSem()

	duration	interrupts disabled
min	?	?
max	?	?

Table 55: OSCreateBinSem() Execution Times

OSCreateMsg()

	duration	interrupts disabled
min	65	55
max	65	55

Table 56: OSCreateMsg() Execution Times

OSCreateMsgQ()

	duration	interrupts disabled
min	?	?
max	?	?

Table 57: OSCreateMsgQ() Execution Times

OSCreateSem()

	duration	interrupts disabled
min	65	55
max	65	55

Table 58: OSCreateSem() Execution Times

OSCreateTask()

	duration	interrupts disabled
min	98	87
max	98	87

Table 59: OSCreateTask() Execution Times

OSInit()

	duration	interrupts disabled
min	10	0
max	10 + OSTASKS x (7 + t_InitTcb) + OSEVENTS x (7 +t_InitEcb)	0

	condition
min	OSCLEAR_GLOBALS is FALSE.
max	

Table 60: OSInit() Execution Times

Note The default for OSCLEAR_GLOBALS is TRUE. Since many compilers automatically zero all uninitialized variables, you may be able to speed up OSInit() by setting OSCLEAR_GLOBALS to FALSE. However, by doing this you will not be able to re-initialize Salvo on-the-fly.

OSSched()

	duration	interrupts disabled
min	14	9
max	118 + t_InsPrioQ	21, 32 + t_InsPrioQ

	condition
min	No eligible task(s).
max	Includes dummy eligible task immediately yielding back to scheduler. Interrupts are enabled while task runs.

Table 61: OSSched() Execution Times

Note The action of the scheduler is divided into two parts. First, the scheduler disables interrupts, gets the most eligible task ready to run, re-enables interrupts and runs it via an indirect call. Second, upon returning to the scheduler an eligible task must be put back into the eligible queue. Interrupts are therefore disabled for two distinct periods in the scheduler.

OSSignalBinSem()

	duration	interrupts disabled
min	?	?
max	? + t_InsPrioQ	? + t_InsPrioQ

	condition
min	No task(s) waiting on binary semaphore.
max	Tasks(s) waiting on binary semaphore.

Table 62: OSSignalBinSem() Execution Times

OSSignalMsg()

	duration	interrupts disabled
min	63	52
max	109 + t_InsPrioQ	98 + t_InsPrioQ

	condition
min	No task(s) waiting on message.
max	Tasks(s) waiting on message.

Table 63: OSSignalMsg() Execution Times

OSSignalMsgQ()

	duration	interrupts disabled
min	?	?
max	? + t_InsPrioQ	? + t_InsPrioQ

	condition
min	No task(s) waiting on message queue.
max	Tasks(s) waiting on message queue.

Table 64: OSSignalMsgQ() Execution Times

OSSignalSem()

	duration	interrupts disabled
min	58	49
max	104 + t_InsPrioQ	95 + t_InsPrioQ

	condition
min	No task(s) waiting on semaphore.

max	Tasks(s) waiting on semaphore.

Table 65: OSSignalSem() Execution Times

OSStartTask()

	duration	interrupts disabled
min	78 + t_InsPrioQ	69 + t_InsPrioQ
max	78 + t_InsPrioQ	69 + t_InsPrioQ

Table 66: OSStartTask Execution Times

OSTimer()

	duration	interrupts disabled
min	23	23
max	57 + t_InsPrioQ	57 + t_InsPrioQ

	condition
min	No task(s) timed out.
max	1 task timed out. Will increase by 47-cycle overhead and each additional task's t_InsPrioQ if/when multiple tasks time out together. E.g. with 2 tasks timing out simultaneously and nothing in the eligible queue, max duration is 57 + 22 + 47 + 57 = 183 cycles.

Table 67: OSTimer() Execution Times

Note `OSTimer()` must be called with interrupts disabled.

Maximum Variable Execution Times

As seen above, the execution times for some Salvo services are a combination of fixed and variable times. Those variable times are dependent on how many tasks are active in your application. As a programmer, it is often useful to know the worst-case execution time for a kernel service. The following section illustrates the worst-case values for Salvo's variable execution times, for 1 to 8 tasks.

Tip The figures in the tables below can be easily extrapolated for systems that use more than 8 tasks. For instance, in Table 8-n the maximum t_InsPrioQ increases by 21 cycles for each additional task.

t_InsPrioQ

The maximum values for t_InsPrioQ for 1 to 8 tasks in test configurations I-V are shown below.

	I	II	III	IV	V
1	22	22	22	22	22
2	57	63	57	63	57
3	78	84	78	84	78
4	99	105	99	105	99
5	120	126	120	126	120
6	141	147	141	147	141
7	162	168	162	168	162
8	183	189	183	189	183

Table 68: Maximum t_InsPrioQ for 1-8 Tasks in Configurations I-V (simple queues)

t_DelPrioQ

The maximum values for t_DelPrioQ for 1 to 8 tasks in test configurations I-V are shown below.

	I	II	III	IV	V
1	36	39	36	39	36
2	53	56	53	56	53
3	70	73	70	73	70
4	87	90	87	90	87
5	104	107	104	107	104
6	121	124	121	124	121
7	138	141	138	141	138
8	155	158	155	158	155

Table 69: Maximum t_DelPrioQ for 1-8 Tasks in Configurations I-V (simple queues)

t_InsDelayQ

The maximum values for t_InsDelayQ for 1 to 8 tasks in test configurations I-V with 8- and 16-bit delays are shown below.

	I	II	III	IV	V
1	n/a	22	n/a	22	22
2	"	82	"	82	77
3	"	105	"	105	100
4	"	128	"	128	123
5	"	151	"	151	146
6	"	174	"	174	169
7	"	197	"	197	192
8	"	220	"	220	215

Table 70: Maximum t_InsDelayQ for 1-8 Tasks in Configurations I - V (simple queues, 8-bit delays, w/OSSPEEDUP_QUEUEING)

	I	II	III	IV	V
1	n/a	22	n/a	22	22
2	"	116	"	116	111
3	"	152	"	152	147
4	"	188	"	188	183
5	"	224	"	224	219
6	"	260	"	260	255
7	"	296	"	296	291
8	"	332	"	332	327

Table 71: Maximum t_InsDelayQ for 1-8 Tasks in Configurations I - V (simple queues, 16-bit delays, w/OSSPEEDUP_QUEUEING)

t_DelDelayQ

The maximum values for t_DelDelayQ for 1 to 8 tasks in test configurations I-V with 8- and 16-bit delays are shown below.

	I	II	III	IV	V
1	n/a	42	n/a	42	38
2	"	59	"	59	55
3	"	76	"	76	72
4	"	93	"	93	89
5	"	110	"	110	106
6	"	127	"	127	123
7	"	144	"	144	140
8	"	161	"	161	157

Table 72: Maximum t_DelDelayQ for 1-8 Tasks in Configurations I - V (simple queues, 8-bit delays)

	I	II	III	IV	V
1	n/a	42	n/a	42	38
2	"	81	"	81	77
3	"	98	"	98	94
4	"	115	"	115	111
5	"	132	"	132	128
6	"	149	"	149	145
7	"	166	"	166	162
8	"	183	"	183	179

Table 73 Maximum t_DelDelayQ for 1-8 Tasks in Configurations I - V (simple queues, 16-bit delays)

See "Impact of Queueing Operations" below for more information on the variable times listed in this section.

Impact of Queueing Operations

Salvo uses queues (linked lists) in RAM to manage tasks and events. The queue elements are task control blocks (tcbs), one for each task in your application. Queues are very efficient at storing information, and by using queues instead of arrays to hold data, Salvo achieves minimal RAM requirements at a cost of increased access times. Since access to queue elements is linear (non-random), much of Salvo's run-time performance is affected by the number of tasks in your application.

All of Salvo's queues are priority queues sorted by a particular tcb field. All queues except the delay queue are sorted by task priority, with the highest-priority task at the head of the queue. The delay queue is sorted by remaining delay, with the task with the shortest delay at the head of the queue.

Most Salvo services involve doing something with a task, either directly or indirectly, and therefore involve queueing operations (insertion or deletion) in one or more queues. That's because each task, unless it is destroyed or stopped, is always in one or more queues, or is in the process of being moved from one queue to another. Therefore the execution times of most services are a combination of fixed times and queueing operations times.

Characterizing the run-time performance of a Salvo service is not as simple as simply stating that "service `OSXyz()` completes in nn instruction cycles." Instead, the execution time for a Salvo service is dependent on how many tasks are in your system, on the priority of the task that the service is operating on, and on the elements in the queue. The fewer the tasks, the faster the service will be. The emptier the queue (in general), the faster the service will be. The higher the task priority (in general), the faster the service will be. The speed of the service may also be dependent on some configuration parameters (e.g. `OSSPEEDUP_QUEUEING`).

Since it's not possible to predict the status of Salvo's queues at the time that a service is called, we'll present the execution times for queueing operations as a function of both the number of tasks of equal or higher priority (compared to the task in question), and the number of tasks of lower priority (compared to the task in question). This is because any or all of the other tasks in your application might be in the queue that task in question is about to enter or leave.

While this may sound complicated, it's actually quite simple. In the example table below, each column represents the number of instructions required for a queueing operation as the number of lower-priority tasks varies and the number of higher-priority tasks is fixed. Each row represents the number of instructions required as the number of higher- or equal-priority tasks varies and the number of lower-priority tasks is fixed.

	0	1	2	3	4	5	n (>0)
0	22	57	78	99	120	141	36 + n*21
1..n	47	68	89	110	131	152	47 + n*21

Table 74: Example of Queueing Operation Times

We can see that the queueing operation will take:

- 22 cycles if there is only 1 task (the queue is empty),

- 120 cycles if there are 4 tasks of higher or equal priority and none of lower,
- 47 cycles if it inserts the highest-priority task and lower-priority tasks exist, and
- 131 cycles if there are 4 higher-priority tasks and one or more lower-priority tasks.

The table provides enough information to extrapolate the queueing operation times to systems with more tasks. In the above example, the execution times for each additional higher-priority task increase by 21 instruction cycles.

There are a variety of ways to interpret these results. For a critical task you could assign it the highest priority (0) and know that no matter what else is happening in your application, the queueing operation will never take more than 47 instruction cycles. You could assign a less important task a priority of 3, and know that the service will never take more than 110 instruction cycles (assuming you assign unique priorities to all of your tasks). Or you could simply note that with 4 tasks running the worst-case execution time is 99 instruction cycles.

Tip To find the worst-case execution time for a queueing operation, scan all the cells in the table whose row and column indexes, when added together, are less than or equal to the number of tasks in your application minus 1.

Note Using the worst-case execution time for a Salvo service as an indication of its run-time performance may be adequate for those cases that are not particularly time-critical. Where the speed of a Salvo service is critical, you may want to adjust the priority of the task in question.

The execution times required to perform queueing operations are presented below. Each parameter is characterized for both simple queues (the Salvo default) and array-based schemes (coming in v3.0). When calculating the total time required for a Salvo service, refer to the appropriate table below to obtain the proper value to add to the service's fixed execution time.

Note Figures for execution times involving the delay queue are based on the number of tasks of equal or lesser remaining delay, and the number of tasks with greater remaining delay. The priority of a task has no effect on its insertion into the delay queue.

Simple Queues

This section lists the times to complete queueing operations when simple queues (the default) are used.

Caution Dedicated test programs are used throughout this chapter to characterize Salvo's real-world performance. Each test program is designed solely to characterize one particular aspect of Salvo's performance, and is often intended for use with an In-Circuit Emulator (ICE) with its support for breakpoints, cycle counts, etc. Because of the unusual design of some test programs, they should not be used as examples of how to write Salvo applications. Refer to the demo and example programs for examples of Salvo application programming using the standard user services.

t_InsPrioQ

t_InsPrioQ is the time to insert an element into a priority queue, and depends on the chosen configuration.

Configurations I & III

	0	1	2	3	4	5	n (>0)
0	22	57	78	99	120	141	36 + n*21
1..n	47	68	89	110	131	152	47 + n*21

Table 75: t_InsPrioQ for Configurations I & III

For simple queues, the time to insert a task into a priority queue depends on the number of higher- or equal-priority tasks in the queue.

Configurations II & IV

	0	1	2	3	4	5	n (>0)
0	22	63	84	105	126	147	42 + n*21
1..n	51	72	93	114	135	156	51 + n*21

Table 76: t_InsPrioQ for Configurations II & IV

Adding support for delays increases t_InsPrioQ slightly.

Configuration V

	0	1	2	3	4	5	n (>0)
0	22	57	78	99	120	141	36 + n*21
1..n	47	68	89	110	131	152	47 + n*21

Table 77: t_InsPrioQ for Configuration V

t_DelPrioQ

t_DelPrioQ is the time to delete an element from a priority queue, and depends on the chosen configuration.

Configurations I & III

	0	1	2	3	4	5	n (>0)
0..n	36	53	70	87	104	121	36 + n*17

Table 78: t_DelPrioQ for Configurations I & III

For simple queues, the time to delete a task from a priority queue depends on the number of higher- or equal-priority tasks in the queue.

Configurations II & IV

	0	1	2	3	4	5	n (>0)
0..n	39	56	73	90	107	124	39 + n*17

Table 79: t_DelPrioQ for Configurations II & IV

Adding support for delays increases t_DelPrioQ slightly.

Configuration V

	0	1	2	3	4	5	n (>0)
0..n	36	53	70	87	104	121	36 + n*17

Table 80: t_DelPrioQ for Configuration V

t_InsDelayQ

t_InsDelayQ is the time to insert an element into the delay queue, and depends on the chosen configuration.

Configurations II & IV

	0	1	2	3	4	5	n (>0)
0	22	75	102	129	156	183	48 + n*27
1..n	80	107	134	161	188	215	80 + n*27

Table 81: t_InsDelayQ for Configurations II & IV and 8-bit delays

	0	1	2	3	4	5	n (>0)
0	22	93	138	183	228	273	48 + n*45
1..n	116	161	206	251	296	341	116 + n*45

Table 82: : t_InsDelayQ for Configurations II & IV and 16-bit delays

For simple queues, the time to insert a task into the delay queue depends on the number of tasks with remaining delays which are less than or equal to the delay of the task being inserted. Insertion times are also affected by the delay size specified in the configuration. A speedup can be obtained by using the `OSSPEEDUP_QUEUEING` configuration option.

	0	1	2	3	4	5	n (>0)
0	22	77	100	123	146	169	54 + n*23
1..n	82	105	128	151	174	197	82 + n*23

Table 83: t_InsDelayQ for Configurations II & IV and 8-bit delays, using OSSPEEDUP_QUEUEING

	0	1	2	3	4	5	n (>0)
0	22	93	129	165	201	237	57 + n*36
1..n	116	152	188	224	260	296	116 + n*36

Table 84: t_InsDelayQ for Configurations II & IV and 16-bit delays, using OSSPEEDUP_QUEUEING

Configuration V

	0	1	2	3	4	5	n (>0)
0	22	70	97	124	151	178	43 + n*27
1..n	75	102	129	156	183	210	75 + n*27

Table 85: t_InsDelayQ for Configuration V and 8-bit delays

	0	1	2	3	4	5	n (>0)
0	22	88	133	178	223	268	43 + n*45
1..n	111	156	201	246	291	336	111 + n*45

Table 86: t_InsDelayQ for Configuration V and 16-bit delays

A speedup can be obtained by using the `OSSPEEDUP_QUEUEING` configuration option.

	0	1	2	3	4	5	n (>0)
0	22	72	95	118	141	164	49 + n*23
1..n	77	100	123	146	169	192	77 + n*23

Table 87: t_InsDelayQ for Configuration V and 8-bit delays, using OSSPEEDUP_QUEUEING

	0	1	2	3	4	5	n (>0)
0	22	88	124	160	196	232	52 + n*36
1..n	111	147	183	219	255	291	111 + n*36

Table 88: t_InsDelayQ for Configuration V and 16-bit delays, using OSSPEEDUP_QUEUEING

t_DelDelayQ

t_DelDelayQ is the time to delete an element from the delay queue, and depends on the chosen configuration.

Configurations II & IV

	0	1	2	3	4	5	n (>0)
0	42	59	76	93	110	127	42 + n*17
1..n	55	72	89	106	123	140	55 + n*17

Table 89: t_DelDelayQ for Configurations II & IV and 8-bit delays

	0	1	2	3	4	5	n (>0)
0	42	59	76	93	110	127	42 + n*17
1..n	64	81	98	115	132	149	64 + n*17

Table 90: t_DelDelayQ for Configurations II & IV and 16-bit delays

Configuration V

	0	1	2	3	4	5	n (>0)
0	38	55	72	89	106	123	38 + n*17
1..n	51	68	85	102	119	136	51 + n*17

Table 91: t_DelDelayQ for Configuration V and 8-bit delays

	0	1	2	3	4	5	n (>0)
0	38	55	72	89	106	123	38 + n*17
1..n	60	77	94	111	128	145	60 + n*17

Table 92: t_DelDelayQ for Configuration V and 16-bit delays

Other Variable-speed Operations

The execution time of the user service `OSInit()` is dependent[151] on the time to initialize the task control blocks and event control blocks in your application. These times are presented below:

t_InitTcb

t_InitTcb is the time to initialize a task control block (tcb).

[151] Assumes `OSCLEAR_GLOBALS` is `TRUE` (default).

Configuration I

duration	interrupts disabled
?	?

Table 93: t_InitTcb for Configuration I

Configuration II

duration	interrupts disabled
?	?

Table 94: t_InitTcb for Configuration II

Configuration III

duration	interrupts disabled
?	?

Table 95: t_InitTcb for Configuration III

Configuration IV

duration	interrupts disabled
?	?

Table 96: t_InitTcb for Configuration III

Configuration V

duration	interrupts disabled
?	?

Table 97: t_InitTcb for Configuration V

t_InitEcb

t_InitTcb is the time to initialize an event control block (ecb).

Configuration I

duration	interrupts disabled
?	?

Table 98: t_InitEcb for Configuration I

Configuration II

duration	interrupts disabled
?	?

Table 99: t_InitEcb for Configuration II

Configuration III

duration	interrupts disabled
?	?

Table 100: t_InitEcb for Configuration III

Configuration IV

duration	interrupts disabled
?	?

Table 101: t_InitEcb for Configuration IV

Configuration V

duration	interrupts disabled
?	?

Table 102: t_InitEcb for Configuration V

Chapter 10 • Porting

Introduction

With its minimal RAM requirements, small code size and high performance, Salvo is an appealing RTOS for use on just about any processor. Even if it hasn't been ported to your processor and/or compiler, you can probably do the port in a few hours – it's pretty simple, since there are usually no stack issues to contend with. You'll have to:

- verify that your compiler can compile the Salvo source code,
- create your own `portuser.h` file with interrupt-controlling macros, a simple context-switcher and a simple label declarator, and
- deal with some other processor-specific issues.

Put another way, you must successfully compile, link and run your Salvo application in its new environment. Below you will find specific instructions and examples to help you port Salvo to another compiler and/or target processor. By examining what was required to port Salvo to very different compilers and processors, you should be able to complete a successful port to your compiler and/or processor, too.

Compiler-Specific Issues in the Source Code

In an effort to improve source code manageability, as many of the compiler-specific issues facing Salvo are contained in the compiler-specific porting files listed below:

Archelon Quadravox AQ430	`portaq430.h`
HI-TECH 8051	`portht51.h`
HI-TECH PICC, PICC Lite and PICC-18	`portpicc.h`

HI-TECH V8C	portv8c.h
IAR MSP430 C	portiar430.h
IAR PIC18 C	portiar18.h
GNU C (x86)	portgcc.h
ImageCraft ICC11	porticc11.h
ImageCraft ICC430	porticc430.h
Keil Cx51	portkc51.h
Microchip MPLAB-C18	portmcc.h
Metrowerks CodeWarrior	portmwcw.h
Mix Power C	portmixc.h

By setting OSCOMPILER to the appropriate value in salvocfg.h, these files are included automatically in salvo.h and address most (but not necessarily all) of the issues you will face when porting Salvo to another compiler and/or target processor. A default file, portuser.h, is provided for you to use as a template when porting to an as-yet-unsupported compiler.

Compiling the Salvo Source Code

As mentioned in *Chapter 1 • Introduction*, Salvo makes extensive use of many C language features. It's not uncommon to find C compilers advertised as being ANSI-compliant which are unable to compile the Salvo source code. Testing the compiler against the Salvo source code first will save you a lot of effort in the long run.

You should start by writing a minimal application to begin the porting process – your salvocfg.h might look like:

```
#define OSCOMPILER            OSUNKNOWN
#define OSENABLE_IDLING_HOOK  TRUE
#define OSTARGET              OSUNKNOWN
#define OSTASKS               1
```

Listing 64: Sample salvocfg.h for Porting Testing

where OSIdlingHook() does something really simple like toggle an output port bit. Your test application can then look like this:

```
#include "salvo.h"
```

```
void OSIdlingHook(void)
{
  /* do something really simple here. */
}

int main( void )
{
  OSInit();

  for (;;)
    OSSched();
}
```

Listing 65: Sample Salvo Application for Porting Testing

One way to test your compiler on this application and on all of the Salvo source code is by setting `OSCOMPILER` to `OSUNDEF` and `OSTARGET` to `OSUNDEF` in `salvocfg.h`[152] and by compiling everything. If the compile fails, and if you determine that it's because your compiler does not support one or more standard C features (e.g. unions, bit fields, pointers to structures, etc.), then you have two options: contact the compiler vendor and find out when the feature will be supported, or choose another compiler. If the compile succeeds, the application still won't run because `portuser.h` contains several empty macros. You must still create various entries in `portuser.h` and address a few other, compiler- and target-specific issues.

With supported compilers, `salvo.h` includes the appropriate compiler-specific file based on the compiler in use. If you are using an as-yet-unsupported compiler, you should copy the blank porting file `portuser.h` from `salvo\inc` to your working directory, and modify it as described below.

Controlling Interrupts

As is commonly the case in an RTOS, Salvo has critical sections of code that require protection from interrupts. Without this protection, Salvo's data structures are likely to become corrupted, your application may hang, and performance may become unpredictable.

Salvo protects critical sections of code via the macros `OSDi()` and `OSEi()`. They simply disable and enable (global) interrupts, re-

[152] This will cause `salvo.h` to `#include "portuser.h"`. Assuming you do not have your own `portuser.h` in any of the compiler's include search paths, `salvo\inc\portuser.h` will be included. This file contains "empty" macro definitions to aid in testing your compiler.

spectively. Many compilers will support the creation of these macros in C (as opposed to assembly language). Below are three examples of how these macros were implemented on different compilers.

HI-TECH PICC Compiler

In the Microchip PIC16C6X family (one of several processors supported by Salvo), a Global Interrupt Enable bit, GIE, is used to control interrupts. This is implemented quite easily in C:

```
#define OSDi() GIE = 0
#define OSEi() GIE = 1
```

PICC compiles each of these macros into just one instruction. This example shows how, in certain instances, an efficient compiler "gives up nothing" relative to direct assembly-language programming.

Mix Power C Compiler

The Power C library already provides functions to enable and disable interrupts. Therefore the required macros are implemented very easily:

```
#define OSDi() disable()
#define OSEi() enable()
```

Metrowerks CodeWarrior Compiler

Currently Salvo does not support calling `OSTimer()` from an interrupt when compiling under CodeWarrior. Therefore there is no need to control interrupts in this case. The macros look like:

```
#define OSDi()
#define OSEi()
```

Context Switching

Creating the context-switching macro `OS_Yield()` is a bit more complicated. This bit of in-line code must save an absolute address via a call to a subroutine and then execute the processor's return-from-subroutine instruction. In pseudo-code, this looks something like:

```
        OSSaveRtnAddr(label);
        return();
label: ;
```

The saving of the address is complicated by the fact that the absolute address is a forward-referenced label pointing to the instruction immediately following the return instruction. Because this construct looks a bit odd to an optimizing C compiler (which might fail to compile all of the code following the `return()` instruction), this macro is usually implemented at least partly in assembly language. See the examples below for in-depth explanations.

HI-TECH PICC Compiler

PICC supports in-line assembly, has a powerful preprocessor, and supports mixed C- and assembly-language programming. But the assembler does not support relative addressing or local assembly-language macro labels, and attempting to define the label in C may result in the compiler reporting undefined symbol errors. A working context switcher is shown below:

```
#define _mkstr(a) #a
#define OS_Yield(label) \
{ \
    OSSaveRtnAddr(label); \
    asm("return"); \
    asm("_" _mkstr(label) ":"); \
}
```

The most difficult issue encountered while creating this macro was getting the label to resolve correctly and without any other warnings or errors from the compiler. A solution was found in defining the label via the assembler. To make an assembler label visible to the compiler, it must be prepended with an underscore, '`_`' and end with a colon, '`:`'. The `_mkstr()` macro is used to properly format the assembly-language label.

Mix Power C Compiler

Power C does not support in-line assembly, only callable functions written in assembly. Also, Power C employs the x86's BP and SP registers in a manner different and incompatible with "standard" IBM C. This means that a totally different approach to a context switcher is required.

A solution was found in creating a context-switching function (as opposed to in-line code) that saved the return address (available on

the x86 stack) directly into the task's tcb, and then popped the stack twice to return all the way from the context-switching function "through" the task up to the scheduler. The function was also complicated by the need to save the x86's segment registers to obtain a complete resume address, and so an additional field had to be added to the task's tcb. The context switcher also detects how the task is entered (for the first time, or subsequently, when it has been resumed).

Metrowerks CodeWarrior Compiler

CodeWarrior supports in-line assembly, but not relative labels. Duplicate labels may cause a compiler exception. By examining the assembly-language output of the compiler for a simple task, a template for the context switcher was derived. The context switcher is implemented as in-line assembly code, and is relatively straightforward:

```
#define OS_Yield(label) \
{ \
  asm {     mov   edx, dword ptr Osctp       }; \
  asm {     mov   dword ptr [edx+0x04], label}; \
  asm {        leave                         }; \
  asm {        ret                           }; \
  asm {label:  push  ebp                     }; \
  asm {        mov   ebp, esp                }; \
}
```

An absolute label is used, and the first two instructions place that address into the second field of the current task's tcb, the task's resume address. A leave instruction is required to clean up the stack, and then the return instruction returns execution to the Salvo scheduler. Upon resuming execution in the task, the base and stack pointers must be initialized, which is done via the last two instructions.

Context-switching Label Declarator

In order to facilitate the declaration of context-switching labels (where necessary), you should implement the `_OSLabel()` macro.

HI-TECH PICC Compiler

Note that the label associated with each instance of `OS_Yield()` is a globally visible label representing an absolute address. In order to successfully compile a Salvo task that has a context switch in it,

an additional declaration per context switch is required to avoid an unidentified identifier compiler error. One way to do this is to declare the label as being a void function, e.g.

```
void Labelname( void );
...
OS_Yield(Labelname);
...
```

Therefore the label declarator is very simple:

```
#define _OSLabel(a) void a(void);
```

It's helpful to use a predictable naming convention in an case like this where Salvo's context-switching macros (i.e. those prefaced by "OS_") require explicit label names. For example, one might adopt a scheme like this:

```
void TaskA1( void );
void TaskA2( void );
void TaskB1( void );
...
void TaskA( void );
{
  ...
  OS_Delay(10, TaskA1);
  ...
  OS_Yield(TaskA2);
}
...
void TaskB( void );
{
  ...
  OS_WaitSem(SEM_1, OSNO_TIMEOUT, TaskB1);
  ...
}
```

Note Salvo's context-switching macros are declared such that an explicit label name, where required, is always the last parameter of the macro.

Mix Power C Compiler

Since this context switcher does not use labels, calls to Salvo's context-switching macros do not actually require the explicit label names:

```
#define _OSLabel(a)
```

However, due to the nature of this context switcher, certain other restrictions apply when compiling Salvo under Power C. See the Power C compiler notes in *Chapter 11 • Tips, Tricks and Trouble-shooting* for further information.

Metrowerks CodeWarrior Compiler

As is the case with HI-TECH PICC, explicit labels are required. However, CodeWarrior treats each instance of an assembly-language label within a function as a local label, and so no declarations are required elsewhere in the code. Therefore the label declarator looks like:

```
#define _OSLabel(a)
```

However, explicit labels may not be repeated inside a function.

Managing RAM

One of the biggest problems you may face while writing code in C for a processor with limited amounts of memory is allocating what little RAM is available to the variables in your application. While this is not a problem with an x86-based processor, it can be a real headache with processors that have banked RAM.

All of Salvo's internal typedefs can be written to be "bank-aware" where necessary. Those that will need bank specifiers are Salvo's *qualified* types. Use the PICC-specific `typedefs` in `portpicc.h` as examples of how to locate Salvo's variables in banked memory.

Pointers

Salvo makes extensive use of pointers. If your target processor is severely resource-limited (e.g. has banked RAM), you'll want to define the pointers that Salvo uses in a manner that minimizes their size. For example, Salvo uses byte-sized pointers to access RAM when compiling under HI-TECH PICC.

If your target processor has plenty of RAM (e.g. a PC), then you can use the default pointer type(s) without much difficulty.

Linking the Salvo Source Code

Once you're able to compile all of the Salvo source code with your compiler, you can be fairly certain that most of the obvious headaches associated with your port are behind you. Unfortunately, the link phase of the port may introduce more subtle problems. These might include:

- errors due to unexpectedly large RAM requirements and
- errors due to interrupt context-saving.

Apart from obvious type mismatches, this is the stage of porting where you are most likely to have to make direct changes to the source code outside of `portuser.h` in order to succeed. Always keep a backup of your work, and follow the `#if OSCOMPILER ==` statements in the source code for hints on how to support your compiler. Try to minimize changing the source code so that you will be able to take advantage of future Salvo revisions without losing your work. A source code control / revision system is very useful when embarking on this stage of the port.

Executing the Salvo Source Code

Finally, once the compile and link phases are error-free, you can try to run your Salvo application. If your test program fails to execute properly, you should try an incremental approach by verifying the run-time operation of one section of Salvo at a time.

A good place to start is to check that Salvo's variables are placed in RAM as you expect, and that `OSInit()` is working properly and is not overwriting other memory locations. Then progress to checking `OSCreateTask()`, `OSSched()`, etc. There are often a handful of issues that make a port difficult, and after they are solved the port remains stable.

Good development tools (in-circuit emulators, background debuggers, etc.) are very useful at this stage of the porting process.

Tip The diversity of compilers and target processors that Salvo supports means that there is lots of helpful information in the source code that can aid you in porting. Don't hesitate to review the source code as part of the porting process.

Once you have your application running, you may find that there are additional areas of optimization possible – some are outlined below.

Optimizing and Saving Registers

Salvo's context-switching scheme is structured such that an optimizing compiler should not affect the proper operation of a context switch. However, if the compiler assumes the persistence of a register's value in code that follows the context switch, instead of reloading the register with the appropriate value, the result may be unpredictable behavior in your application.

You have several options as a means of correcting this problem. First, you in your application you could write code (in C) to ensure that the register(s) is(are) reloaded with the appropriate variable(s). Second, you could simply disable optimization around the context switch(es). Lastly, you could expand the operation of the context switcher to save registers prior to returning to the scheduler, and then restore them upon resuming the task. You would need to edit the Salvo source code to modify the context switcher and to add storage locations to the task control block structure in order to do this.

Verifying the Required Stack Depth

Salvo requires a minimum call ... return stack depth. You'll need to verify that your target processor has a hardware stack at least this deep in order to be able to run a Salvo application. Note that this stack depth depends on how your compiler implements indirect function calls in C (see `OSSched()` in `sched.c`).

Characterizing RAM and ROM Requirements

Once your compiler is successfully compiling all of the Salvo source code, you can get a basic idea of how efficient it is in terms of RAM and ROM usage by compiling a simple Salvo application. A good place to start would be one of the demo or test programs included with the Salvo distribution.

Characterizing Run-time Performance

For hard real-time systems, where it's critical that one or more operations be performed within a specified time frame or number of cycles, you will want to know exactly how long a particular operation takes. For example, you may want to know the worst-case execution time for a particular Salvo service.

It's no exaggeration that the better your tools, the easier the work ahead of you. Ideally you'll want in in-circuit emulator with programmable breakpoints and a cycle counter. With it, you can zero the cycle counter at the entry to a Salvo service, and break at the exit to obtain the total number of execution cycles, and hence the elapsed time. You may also be able to do this in a simulator.

The programs found in `salvo\test` are used to characterize Salvo's run-time performance. Once you are able to compile and run a Salvo application with your compiler, you should be capable of using the test programs to characterize Salvo's run-time performance in your system. See Chapter 9 • Performance for more information.

Chapter 11 • Tips, Tricks and Troubleshooting

Introduction

If you're having trouble getting your code to work properly with Salvo, here are some suggestions on how to solve your problem.

- Read and re-read all the relevant portions of this manual.
- Review the example programs in this manual and in the Salvo distribution. You may find something that is very similar to what you are trying to do.
- Examine the postprocessed output of your compiler, both in C and in assembly language. Output listings contain a wealth of useful information.
- Examine any map files generated by your compiler. These files have information containing the location of Salvo routines and variables and their sizes, the calling trees, etc.
- Use the error codes returned by the user services to verify that the desired Salvo actions are really happening.
- If your application has the RAM and ROM to support it, use `OSRpt()` to examine the status of the system.
- If you have access to run-time debugging tools, step through the code in question while monitoring important variables.
- Examine the Salvo source code – it may contain information not presented elsewhere.

Most importantly, examine your assumptions! Don't assume, for example, that a call to `OSStartTask()` is working until you've confirmed that it is in fact returning an error code of `OSNOERR`.

Compile-Time Troubleshooting

I'm just starting, and I'm getting lots of errors.

Be sure to place

```
#include <salvo.h>
```

at the start of each source file that uses Salvo.

My compiler can't find salvo.h.

Make sure that your compiler's include search paths contain the `salvo\inc` directory.

My compiler can't find salvocfg.h.

Each project needs a project-specific `salvocfg.h`. Create one from scratch or copy one from another project. `salvocfg.h` normally resides in your current working directory – you may need to instruct your compiler to explicitly search this directory.

If you are using a Salvo freeware library, copy its `salvocfg.h` to your working directory and edit it as needed.

My compiler can't find certain target-specific header files.

This problem may arise if your compiler has no generic target processor header file that uses defined symbols to include the appropriate target-specific header file. The solution is to include the target-specific header file in your `salvocfg.h`.

My compiler can't locate a particular Salvo service.

You must either include the Salvo files in your project or link to a Salvo library. If you are including files, you'll need to add the `salvo\src` directory to the include search path.

If you are linking to libraries, you'll need to add the library's path to the library search path. You may also need to add `salvo\src` to the include search path because of the need to compile modules like `mem.c` along with your application.

See Application Note *AN-1 Using Salvo Freeware Libraries with the HI-TECH PICC Compiler* for a complete example of adding search paths to a project.

My compiler has issued an "undefined symbol" error for a context-switching label that I've defined properly.

This may be happening if you have the context-switching label in unreachable code and your compiler has removed the unreachable code through optimization. For example, `OS_Delay()` below is unreachable because of an innocuous error:

```
if ( speed = 0 )
  outPWM = 0;
else {
  outPWM = 1;
  OS_Delay(speed, label);
  …
}
```

and your compiler may be unable to find label as a result. Change your code to make the context switch reachable[153] and the error should disappear.

My compiler is saying something about OSIdlingHook.

The configuration options in your `salvocfg.h` may be set to enable the user hook function, `OSIdlingHook()`. You must define a function with this name. For example,

```
void OSIdlingHook(void)
{
  ;
}
```

is a null (i.e. "do-nothing") function that satisfies this requirement.

My compiler has no command-line tools. Can I still build a library?

You can build a library without access to a command-line librarian[154] by creating a project with all of the Salvo source files, and

[153] Use `if ( speed = = 0 )` instead of `if ( speed = 0 )`.

setting the output type of your compiler to be a library file. You will also need a special `salvocfg.h` file that looks like this:

```
#define OSUSE_LIBRARY      TRUE

#define OSLIBRARY_TYPE     OSL
#define OSLIBRARY_CONFIG   OST
#define OSLIBRARY_VARIANT  OSNONE

#undef  OSMAKE_LIBRARY
#define OSMAKE_LIBRARY     TRUE
```

This works as follows: when you set `OSUSE_LIBRARY` to `TRUE` in your project's header file `salvocfg.h`, the library header file `salvolib.h` will be included in your project. By defining the library type, configuration and variant symbols `T`, `C` and `V`, respectively, and by setting `OSMAKE_LIBRARY` to `TRUE`, the Salvo source code is configured for library building.

This method is inefficient for building multiple libraries. For that, refer to Salvo's makefiles.

Run-Time Troubleshooting

Nothing's happening.

Did you remember to:

- Call `OSInit()`?
- Set `OSCOMPILER`, `OSTARGET` and `OSTASKS` correctly in your `salvocfg.h`?
- Create at least one task with `OSCreateTask()`?
- Choose valid task pointers and task priorities that are within the allowed range?
- Call the Salvo scheduler `OSSched()` from inside an infinite loop?
- Task-switch inside each task body with a call to `OS_Yield()`, `OS_Delay()`, `OS_WaitXyz()` or another context-switcher?
- Structure each task with its body in an infinite loop?

154 CodeWarrior v3.1 has no command-line tools, but can build a library from a project.

If you've done all these things and your application still doesn't appear to work, you may have a configuration problem (e.g. parts of your `salvocfg.h` do not match those used to create the freeware library you're using) or an altogether different problem.

Also, make sure that you've done a full recompile ("re-make"), and, if you're using some sort of integrated development environment, be sure that you've downloaded your latest compiled code and reset the processor before running the new code.

It only works if I single-step through my program.

This is usually indicative of a problem with interrupts or the watchdog timer. Since both are usually disabled when single-stepping with an in-circuit emulator (ICE) or in-circuit debugger (ICD), your application may work in this mode but not in run mode.

If your application uses interrupts, be sure that any *interrupt flags* are cleared before leaving the ISR. When interrupt sources share the same interrupt vector, failing to clear the interrupt flag will result in an endless loop of interrupt services. In general, vectored interrupts do not have interrupt flags associated with them.

Many target processors enable the watchdog timer by default. If you fail to reset it regularly, your application will appear to be constantly resetting itself. Depending on the watchdog timer's timeout period, this may be a very short (e.g. < 1s) period. Either disable the watchdog timer or use Salvo's `OSCLEAR_WATCHDOG_TIMER()` configuration option.

Note All Salvo projects in the distributions are compiled with `OSCLEAR_WATCHDOG_TIMER()` defined to reset the watchdog timer. This way, even if you forget to disable the watchdog timer[155] in your development environment, the application should still work.

It still doesn't work. How should I begin debugging?

If you have the ability to set breakpoints, a quick way to verify that your application is multitasking is to re-load your executable (e.g. hex) code, place breakpoints at the entry of each task, reset the

---

[155] In the Microchip development tools family, the PICMASTER and the MPLAB-ICE disable the watchdog timer by default, but the MPLAB-ICD enables it by default.

processor, and Run. If you have successfully initialized Salvo and created tasks (check the error return codes for `OSInit()` and `OSCreateTask()`), the first call to `OSSched()` should eventually result in the processor halting at one of those breakpoints.

If your application makes it this far, Salvo's internals are probably working correctly, and your problem may have to do with improper task structure and/or use of Salvo's context-switching services. Improper control of interrupts and incorrectly-written interrupt service routines (ISRs) are also a common problem.

If you do not have hardware debugging support, use simple methods (like turning an LED on or off from within a task) to trace a path through your program's execution. On small, embedded systems, "`printf`-style debugging" may not be a viable option, or may introduce other errors (like stack overflow) that will only frustrate your attempts to get at the root of the problem.

My program's behavior still doesn't make any sense.

You may be experiencing unintended interaction with your processor's watchdog timer. This can occur if you've compiled your application with the target processor's default (programmable) configuration, which may enable the watchdog timer. You can avoid this problem by using the `OSCLEAR_WATCHDOG_TIMER()` configuration option in your `salvocfg.h` configuration file. By defining this configuration option to be your target processor's watchdog-clearing instruction, the Salvo scheduler will clear the watchdog each time it's called, and prevent watchdog timeouts.

Compiler Issues

Where can I get a free C compiler?

Borland's C++ compilers can be had for free at:

http://www.borland.com/bcppbuilder/freecompiler/

They can be used to create 16- and 32-bit PC (x86) applications.

HI-TECH software also offers free C compilers:

http://www.htsoft.com/

Pacific C can be used to create PC (x86) applications, and PICC Lite can be used on the Microchip PIC16C84 family.

Where can I get a free make utility?

You can download the GNU make utility's source code from

http://www.gnu.org/order/ftp.html

A precompiled DOS/Win32 version is available at

ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2gnu/

Look for the `mak*.zip` files. This is a full-featured, UNIX-like make that works well in the Win32 environment.

Where can I get a Linux/Unix-like shell for my Windows PC?

You can download the Cygwin bash shell from RedHat at

http://sources.redhat.com/cygwin/

A full installation will contain GNU make and many other utilities. It works best on Windows NT / 2000 / XP systems. If you have the full version of Salvo, this shell can be used to generate the standard and freeware libraries on a Windows PC.

My compiler behaves strangely when I'm compiling from the DOS command line, e.g. "This program has performed an illegal operation and will be terminated."

The DOS command line is limited to a maximum of 126 characters. If you invoke your compiler with a longer command line, you may experience very unpredictable results. The solution is to reorganize your project. Consult your compiler's user's manual for more information.

Another possibility is that the environment size on your Windows/DOS PC is inadequate for the DOS program(s) you are running. If you run more than one DOS window under Windows and the environment size is marginal, you may also encounter this

problem. You can fix this by adding the shell command to your config.sys file, e.g.:

```
shell = c:\windows\command.com /p /e:nnnnn
```

where nnnnn is the size of the environment, in bytes, from 160 to 32768. The default is 256. See your DOS manual for more information on the DOS command interpreter and the shell command.

My compiler is issuing redeclaration errors when I compile my program with Salvo's source files.

If you create your application by compiling and then linking your files and Salvo's source files all at once, be sure that none of your source files have the same name as any Salvo source file.

HI-TECH PICC Compiler

Salvo has been thoroughly tested with PICC and it is unlikely that you will encounter any problems that are due directly to compiling and linking the Salvo code to your application. However, since it is often difficult to pinpoint the exact cause of a compile-and-link error, you should follow the tips below if you encounter difficulties.

Running HPDPIC under Windows 2000 Pro

Some people like to run HPDPIC[156] in an 80x50 "DOS window" under Windows. Do the following:

- start HPDPIC
- right-click on the menu bar and select Properties
- select Layout
- choose a Window Size of Width:80 and Height:50
- select OK, choose "Save properties for future windows with same title", select OK
- exit HPDPIC (alt-Q)
- restart HPDPIC

You may want to choose a different font or font size (under Properties->Font) that is better suited to a larger DOS window. If you are having problems with your mouse, instead of changing the

156 The HI-TECH Integrated Development Environment (IDE) for PICC.

window size settings in the procedure above, deselect the Quick-Edit mode under Properties->Options.

Setting PICC Error/Warning Format under Windows 2000 Pro

In Windows 2000 Pro, do either:

My Computer > Properties > Advanced >
Environment Variables ...

or

Start > Settings > Control Panel > System >
Advanced > Environment Variables ...

then in User Variables for *Userid* do:

New > Variable, enter HTC_ERR_FORMAT , OK,
Variable Value, enter Error[ ] %f %l : %s , OK

and

New > Variable, enter HTC_WARN_FORMAT , OK,
Variable Value, enter Warning[ ] %f %l : %s , OK

Then log off and log back on for these changes to take effect. You can see that they are in force by running the MS-DOS Prompt (`C:\WINNT\system32\command.com`) and entering the `SET` command. Type `EXIT` to leave the MS-DOS command prompt.

Note that you must log off and log back on for these changes to take effect. If you change the environment variables without logging off and back on, MPLAB may behave strangely, like do nothing when you click on the error/warning message.

Linker reports fixup errors

If the PICC linker is unable to place variables in RAM, it will report fixup errors. Interpreting these errors can be very difficult. You must successfully place all variables in RAM before attempting to interpret any other PICC link errors. If you're having difficulty, the simplest thing is to place all of Salvo's variables in an unused bank (e.g. Bank 3 on a PIC16C77). Then, by using PICC's bank directives you can move your own variables around until they all fit. A thorough understanding of the bank directives is required,

especially when banked (or unbanked) pointers to banked (or unbanked) objects are involved. Consult the PICC manual for more information, or the Salvo source code for examples of using the bank directives.

See also "Placing Variables in RAM", below.

Placing variables in RAM

Because PICs have generally very little RAM, as your application grows it's likely that you will need to explicitly manage where variables are located in RAM. If your Salvo application has more than a few tasks and events, it's likely that you will want to place the Salvo data structures (e.g. tcbs and ecbs) and other variables in a RAM memory bank other than Bank 0, the default bank for auto variables and parameters. To do this, use the `OSLOC_Xyz` configuration options and recompile your code. The `OSLOC_Xyz` configuration words options not all be the same – for example you can place ecbs in Bank 2, and tcbs in Bank 3.

If you need to use more than one bank to place Salvo's variables in RAM, for best performance place them in bank pairs – e.g. in Banks 2 and 3 only.

Note Your Salvo code will be smallest if you place all of your Salvo variables in Bank 1 and/or Bank 0. PICC places all auto variables in Bank 0. Bank switching is minimized by placing Salvo's variables in the same bank as the auto variables.

Link errors when working with libraries

If you get the following error:

```
HLINK.EXE::Can't open (error): : No such file or
directory
```

while working with multiple projects and libraries, it may go away be simply re-making the project.

Avoiding absolute file pathnames

Use HPDPIC's Abs/Rel path feature when adding source and include files to your project. You'll be able to enter path names much more quickly.

Compiled code doesn't work

Make sure you're using the latest version of PICC, including any patches that are available. Check http://www.htsoft.com for version updates.

PIC17CXXX pointer passing bugs

On the 17C756, in certain cases PICC failed to correctly dereference pointers passed as parameters. This affected Salvo's queueing routines.

Note This was fixed in PICC v7.84.

While() statements and context switches

You may encounter a subtle problem if you use a `while()` statement immediately following a Salvo context switch, e.g.

```
...
OS_Delay(5);
while ( rxCount ) {
...
```

if rxCount is a banked variable, after optimization the compiler may fail to set the register page bits properly when accessing the variable. This will probably lead to incorrect results. A simple workaround is to add the line

```
rxCount = rxCount;
```

between the context switch and the `while()` statement. This will "force" the proper RP bits.

Note This was fixed in PICC v7.85.

Library generation in HPDPIC

If you are using HPDPIC projects to compile libraries for use with PIC processors with different numbers of ROM and RAM banks (e.g. PIC16C61 and PIC16C77), you may encounter an error when linking your application(s) to one of those libraries. This is because the PICC preprocessor `CPP.EXE` may be fed the wrong processor-selection argument if you're switching between projects with different processors.

The solution is to first load a project whose output is a `.COD` file, and then load a second project destined for the same type of processor and whose output is a library. Make the library (i.e. make the second project), then re-load the first project, and make it, linking to the previously generated library. By loading the first project you correctly set the processor type for the second project.

Note This was fixed in PICC v7.86.

Problems banking Salvo variables on 12-bit devices

On the 12-bit devices (e.g. PIC16C57), Salvo applications don't work when Salvo variables are placed in a RAM bank other than Bank 0. The solution is to upgrade to the latest version of the compiler.

Note This was fixed in PICC v7.86PL4.

Working with Salvo messages

Salvo messages are passed via void pointers. Use the predefined type definition (`typedef`) `OStypeMsgP` when declaring pointers to messages. This type is defined by default as `void *`. In PICC a pointer to a void object points only to RAM. That's fine if your Salvo application has only messages in RAM. But what if you want to send messages which point to objects in ROM (e.g. a string like "STOP" or "GO") as well as RAM? By changing `OSBIG_MESSAGE_POINTERS` to `TRUE` messages can now point to objects in RAM or ROM. This may add 1 extra byte to the size of each event control block (ecb).

Note `OSBIG_MESSAGE_POINTERS` must be set to `TRUE` in your `salvocfg.h` if you are using messages and/or message queues and you are accessing message data that's in ROM.

See also *Working with Message Pointers* in this chapter.

Adding OSTimer() to an Interrupt Service Routine

If you are linking to a freeware or custom Salvo library, or if `timer.c` is one of the nodes in your project, and you call `OSTimer()` from within an interrupt routine, PICC automatically

assumes the worst case with regard to register usage within `OSTimer()` and the functions it may call, and automatically adds a large number of register save and restores to your interrupt routine. This makes it large and slow, which is undesirable.

The solution is to change the organization of your source files. Instead of compiling `timer.c` into a linkable object module, *include it in your source file* which contains the call to `OSTimer()`. For example, your `main.c` might now look like this:

```
…
#include "timer.c"

void interrupt intVector( void )
{
  /* handle various interrupts         */
  ...

 /* this happens every 10ms.           */
  if ( TMR1IF ) {
    /* must clear TMR2 interrupt flag. */
    TMR1IF = 0;

    /* reload TMR1 while it's stopped. */
    TMR1ON = 0;
    TMR1 -= TMR1_RELOAD;
    TMR1ON = 1;

    OSTimer();
  }
}
```

By including `timer.c` in the same source code file as the interrupt routine, PICC is able to deduce exactly which temporary registers must be saved when the interrupt occurs and restored thereafter, instead of assuming the worst case and saving and restoring all of them. The resultant savings in code space and improvement in interrupt execution speed are substantial. If your application uses the Salvo timer, this reorganization is highly recommended.

After including `timer.c` in your interrupt source code file, you may want to recompile your custom Salvo library if you are using one. The Salvo functions will still be able to reference the required queueing functions – they've simply moved from the library to your object modules.

Note You may need to add the switch `-I\salvo\src` to PICC's command line in order for the compiler and linker to find the `timer.c` source file.

Using the interrupt_level pragma

Whenever you call any Salvo services from both inside an interrupt and from background code (e.g. from within a task), you must insert the following PICC directive prior to your interrupt routine:

```
#pragma interrupt_level 0
```

This alerts the PICC compiler to look for multiple call graphs of functions called from both mainline and interrupt code. This is necessary in order to preserve parameters and auto variables.

Note Placing this PICC pragma before an interrupt routine has no deleterious effects even when multiple call graphs are not generated. Therefore it's recommended that you always do this if you call any functions from within your interrupt routine.

HI-TECH V8C Compiler

The initial Salvo port to the VAutomation V8-µRISC™ requires an updated V8 assembler, `ht-v8\bin\asv8.exe`, dated 6-21-2001 or later, along with v7.84 of the compiler. Many of the test programs (e.g. `salvo\test\t41\sysl`) use `printf()` for run-time output for use with the simulators.

Note Since the HI-TECH V8C compiler and its HPDV8 IDE are substantially similar in operation to HI-TECH's PICC compilers and HPDPIC IDE, refer to *HI-TECH PICC Compiler*, above, for related information.

Simulators

Two simulators for the V8-µRISC™ are available – one from HI-TECH (`simv8.exe`) and one from VAutomation (`v8sim.exe`). Salvo applications run on both.

HI-TECH 8051C Compiler

Problems with static initialization and small and medium memory models.

When using the small or medium memory models, the compiler issues the error `Can't generate code for this expression` when faced with the declaration

```
unsigned int counter = 0;
```

This occurs because initialized objects are in ROM for these models, and therefore cannot be changed. The solution is to either declare the variable as near, or explicitly initialize it elsewhere in your code.

IAR PICC Compiler

Target-specific header files

The IAR PICC compiler requires a target-specific header file that contains symbols and addresses for the PICmicro special function registers (SFRs). These files are located in the `inc` subdirectory of the compiler's distribution, and are target-specific.

For example, `\iar\ew23\picmicro\inc\io17c756.h` is the header file for the 17C756 PICmicro. By placing

```
#include "io17C756.h"
```

in your `salvocfg.h` file,[157] the compiler will be able to correctly resolve certain symbols used throughout the Salvo source code.

Interrupts

The vector for each interrupt must be properly defined. Use the compiler's vector pragma like this:

```
#pragma vector=0x10
__interrupt void intVector(void)
{
```

[157] As opposed to placing it in your `main.c`.

```
    T0IF = 0;
    TMR0 -= TMR0_RELOAD;
    OSTimer();
  }
```

This will place the TMR0 interrupt vector at 0x10 on a PIC17C756.

Mix Power C Compiler

In contrast to usual IBM C call stack programming, which has positive offsets from BP for function arguments and negative offsets from BP for local variables, the Power C compiler uses positive offsets from BP to access both local variables and function arguments. This affects the Salvo context switcher for Power C to the degree that it will only function correctly as long as the call stack for the task is in its simplest form. The key to compiling Salvo applications to run on the PC is to guarantee that each task has the simplest possible Power C entry call stack.

Strict adherence to the Salvo requirement that only static local variables be used in a task is required to avoid run-time errors. Additionally, there are a few other innocuous things ("gotchas") that the Power C programmer might do which violate Salvo's requirement that the call stack remain in its simplest form. Those that are known are outlined below.

Required compile options

When compiling Salvo source code, using the following compile options for `PC.EXE`:

```
/r-
/2
/mm
```

Failure to use these options or to use other incompatible options may prevent your Salvo executable from running properly.

Below is an example line from a makefile:

```
PCopts = /c /o /w /r- /2 /mm /id:\salvo\inc
```

Application crashes after adding long C source lines to a Salvo task

If you have source code (e.g. a function with multiple parameters) within a task that is too long to fit on a single line, you must use the '\' character to continue on the next line, even if it's not necessary for a successful compile. This is because Mix Power C changes the task's entry call stack to one that is incompatible with Salvo's context switcher if the line is not continued with the '\' character. For example, the call to `DispLCD()` below

```
void TaskMsg ( void )
{
  for (;;){
    ...
    DispLCD((char *) ((t_dispMsg *)msgP)->strTop,
      (char *) ((t_dispMsg *)msgP)->strBot);
    OS_Delay((OStypeDelay) ((t_dispMsg *)msgP)-
>delay);
    ...
  }
}
```

will compile successfully, but it will cause the PC application to crash when it runs `TaskMsg()`. By adding the '\' character to the DispLCD() line. e.g.

```
DispLCD((char *) ((t_dispMsg *)msgP)->strTop, \
  (char *) ((t_dispMsg *)msgP)->strBot);
```

the problem is resolved.

Application crashes after adding complex expressions to a Salvo task

Mix Power C changes the task's entry call stack if the expressions in a task exceed a certain level of complexity. For example, placing either

```
char = RxQ[rxHead++];
```

or

```
(dummy = dummy);
```

inside a task will cause problems, whereas replacing them with

```
char = RxQ[rxHead];
rxHead++;
```

and

```
dummy = dummy;
```

will not.

Application crashes when compiling with /t option

Mix Power C changes the task's call entry stack when trace information for the debugger is enabled via the compiler's /t option. This change is incompatible with Salvo's context switcher for Power C. Source code modules which contain Salvo tasks must not be compiled with the /t option.

One way around this problem is to move functionality that does not involve context switching out of the module the task is in and into a separate source code module, and call it as an external function from within the task. A module that does not contain any Salvo tasks can be compiled with the /t option, and hence debugged using Mix Power Ctrace debugger.

Compiler crashes when using a make system

Make absolutely sure that your DOS command line does not exceed 127 characters in length. If it does, the results can be very unpredictable. Simplify your directory structure to minimize pathname lengths when invoking any of the Mix Power C executables (e.g. `PCL.EXE`).

Metrowerks CodeWarrior Compiler

Compiler has a fatal internal error when compiling your source code

Ensure that you do no use duplicate labels in any single source code file. This may occur unintentionally if you duplicate labels for Salvo context-switching macros inside a single function. For example,

```
void Task1( void )
{
  ...
  OS_Delay(1, here);
```

```
  ...
}

void TaskB( void )
{
  ...
  OS_Delay(1, here);
  ...
  OS_Yield(here);
  ...
}
```

may cause a CodeWarrior exception because of the duplicate label a in `Task2()`, whereas

```
void Task1( void )
{
  ...
  OS_Delay(1, here);
  ...
}

void Task2( void )
{
  ...
  OS_Delay(1, here);
  ...
  OS_Yield(there);
  ...
}
```

may not.

Microchip MPLAB

The Stack window shows nested interrupts

The MPLAB Stack window cannot differentiate between an interrupt and an indirect function call. Because Salvo makes extensive use of indirect function calls, you may be seeing a combination of return addresses associated with interrupts and indirect function call return addresses.

Controlling the Size of your Application

The Salvo source code is contained in several files and is comprised of a large body of functions. Your application is unlikely to use them all. If you compile and link the Salvo source files along

with your application's source files to form an executable program, you may inadvertently end up with many unneeded Salvo functions in your application. This may prevent you from fitting your application into the ROM of your target processor.

The solution is to compile the Salvo source files separately, and combine them into a single library. You can then link your application to this library in order to resolve all the external Salvo references. Your compiler should extract only those functions that your application actually uses in creating your executable application, thus minimizing its size.

You must always recreate the Salvo library in its entirety whenever you change any of its configuration options.

Refer to your compiler's documentation on how to create libraries from source files, and how to link to those libraries when creating an executable.

See *Chapter 4 • Tutorial* for more information on compiling your Salvo application.

Working with Message Pointers

If you want to use messages as a means of intertask communications, you'll have to be comfortable using Salvo message pointers. Salvo provides predefined type definitions (C `typedefs`) for working with message pointers. The following message pointer declarations are equivalent:

```
OStypeMsg * messagePointer;
```

and

```
OStypeMsgP messagePointer;
```

but you should always use the latter to declare local or global message pointer variables, both static and auto.

In general, Salvo message pointers are of type `void *`. However, you should use the predefined types to avoid problems when a void pointer is not correct for a message pointer. This occurs mainly with processors that have banked RAM.

When passing an object that is not already a message pointer, you'll need to typecast the object to a message pointer in order to

avoid a compiler error. The following two calls to `OSSignalMsg()` are equivalent:

```
OSSignalMsg(MSG1_P, (OStypeMsg *) 1);
```

and

```
OSSignalMsg(MSG1_P, (OStypeMsgP) 1);
```

The typecast above is required because 1 is a constant, not a message pointer. Here are some more examples of passing objects that are not message pointers:

```
char letter = 'c';
OSSignalMsg(MSG_CHAR_VAR_P, (OStypeMsgP) &letter);

const char CARET = '^';
OSSignalMsg(MSG_CHAR_CONST_P, (OStypeMsgP)
&CARET);

unsigned int * ptr;
OSSignalMsg(MSG_UINT_P, (OStypeMsgP) ptr);

void Function(void);
OSSignalMsg(MSG_FN_P, (OStypeMsgP) Function);
```

Once an object has been successfully passed via a message, you will probably want to extract the object from the message via `OS_WaitMsg()`.[158] When a task successfully waits a message, Salvo copies the message pointer to a local message pointer (`msgP` below) of type `OStypeMsgP`. To use the contents of the message, you'll need to properly typecast and dereference it. For the examples above, we have:

```
char:            * (char *) msgP

const char:      * (const char *) msgP

unsigned int *:    (unsigned int *) msgP

void * (void):     (void * (void)) msgP
```

Failing to properly typecast an object (e.g. using `(char *)` instead of `(const char *)` when dereferencing a constant) will have unpredictable results. Please see Application Note *AN-3 Salvo, Banked Objects and the HI-TECH PICC Compiler* for more information on dereferencing pointers.

---

158 An exception occurs when you are not interested in the contents of the message, but only that it has arrived.

NOTE When working with message pointers, it's very important to ensure that Salvo's message pointer type `OStypeMsgP` is properly configured for the kinds of messages you wish to use. On most targets, the default configuration of `void *` will suffice … but there are some exceptions.

For example, the HI-TECH PICC compiler requires 16 bits for `const char` pointers, but only 8 bits for `char` pointers. Therefore the Salvo code (whether in a library or in a source-code build) must be configured to handle these larger pointers or else you will encounter runtime errors.

Appendix A • Recommended Reading

Salvo Publications

A variety of additional Salvo publications are available to aid you in using Salvo. Where applicable, some are included in certain Salvo distributions.

Application Notes

AN-1 Using Salvo Freeware Libraries with the HI-TECH PICC Compiler

AN-2 Understanding Changes in Salvo Code Size for Different PICmicro Devices

AN-3 Salvo, Banked Objects and the HI-TECH PICC Compiler

AN-4 Building a Salvo Application with HI-TECH PICC and Microchip MPLAB

AN-5 Using Salvo with Microchip MPLAB-ICD

AN-6 Designing a Low-Cost Multifunction PIC12C509A-based Remote Fan Controller with Salvo

AN-7 Ninety-Day Countdown Timer Uses Salvo's Delay Services

AN-8 Implementing Quad 1200 baud Full-Duplex Software UARTs with Salvo

Assembly Guides

AG-1 Assembling the SSDL/SCU PICmicro Protoboard

AG-5 Assembling the SSDL/SCU PIC17 Protoboard

Learning C

K&R

Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, New Jersey, 1978, ISBN 0-13-110163-3.

Of Interest This book is the definitive, original reference for the C programming language.

C, A Reference Manual

Harbison, Samuel P. and Steele, Guy L., Jr., *C, A Reference Manual*, Prentice-Hall, NJ, 1995, ISBN 0-13-326224-3.

Of Interest A modern C language reference.

Power C

Mix Software, *Power C, The High-Performance C Compiler*, 1993.

Of Interest Mix Power C is a very inexpensive, full-featured ANSI-compatible C compiler for use on the PC. Its excellent 600+-page manual contains comprehensive tutorial and reference sections. Library source code is available.

Real-time Kernels

µC/OS & MicroC/OS-II

Labrosse, Jean J., *µC/OS, The Real-Time Kernel*, R&D Publications, Lawrence, Kansas, 1992, ISBN 0-87930-444-8.

Labrosse, Jean J., *MicroC/OS-II, The Real-Time Kernel*, R&D Books, Lawrence, Kansas, 1999, ISBN 0-87930-543-6.

Of Interest This book and its greatly expanded and well-illustrated successor provide an excellent guide to understanding

RTOS internals. It also demonstrates how even a relatively simple conventional RTOS requires vastly more memory than Salvo. Its task and event management is array-based. Source code is included.

CTask

Wagner, Thomas, *CTask, A Multitasking Kernel for C*, public domain software, version 2.2, 1990, available for download on the Internet.

Of Interest The author of this well-documented kernel takes a very hands-on approach to describing its internal workings. CTask is geared primarily towards use on the PC. As such, it is *not* a real-time kernel. Its task and event management is primarily queue-based. Source code is included.

Embedded Programming

Labrosse, Jean J., *Embedded Systems Building Blocks*, R&D Publications, Lawrence, Kansas, 1995, ISBN 0-13-359779-2.

Of Interest This book provides canned routines in C for a variety of operations (e.g. keypad scanning, serial communications and LCD drivers) commonly encountered in embedded systems programming. RTOS- and non-RTOS-based approaches are covered. The author also provides an excellent bibliography. Source code is included.

LaVerne, David, *C in Embedded Systems and the Microcontroller World*, National Semiconductor Application Note 587, March 1989, http://www.national.com.

Of Interest The author's comments on the virtues of C programming in embedded systems are no less valid today than they were in 1989.

RTOS Issues

Priority Inversions

Kalinsky, David, "Mutexes Prevent Priority Inversions," *Embedded Systems Programming*, Vol. 11 No. 8, August 1998, pp.76-81.

Of Interest An interesting way of solving the priority inversion problem.

Microcontrollers

PIC16

Microchip, *Microchip PIC16C6X Data Sheet*, Section 13.5, Interrupts, 1996.

Of Interest A special method for disabling the global interrupt bit GIE is required on the PIC16C61/62/64/65. Set `OSPIC16_GIE_BUG` to `TRUE` when using these and certain other processors. The later versions (e.g. PIC16C65A) do not require this fix. Below is a response from Microchip to a customer query on this issue:

The GIE issue is not a 'bug' in the part it relates more to an operational consideration when the GIE bit is handled in software to disable the interrupt system and the fact that during execution of that operation it is possible for an interrupt to occur. The nature of the MCU core operation means that whilst the current instruction is flowing through the device an asynchronous interrupt can occur. The result of this is that the processor will vector to the ISR disable GIE, handle the Interrupt and then enable GIE again. The result of this is of course that the instruction to disable GIE has been overridden by the processor vectoring to the interrupt and disabling then enabling the interrupt. This is a very real possibility and AN576 is explaining a method to ensure that, in the specific instance where you wish to disable GIE in software during normal execution that your operation has not been negated by the very action you wish to stop.

The app note is related to the disabling of GIE in software. The disabling and re-enabling of GIE when an interrupt occurs is performed in hardware by the processor and the execution of the RETFIE instruction. The GIE check is a safeguard to ensure your expected/desired operation has occurred and your program can then operate as expected/desired without the unexpected occurrence of an interrupt. This issue remains on the current range of parts since it is related to the operation of the core when the user wishes to take control of the interrupt system again.

BestRegards,

UK Techhelp

Appendix B • Other Resources

Web Links to Other Resources

Here are some web sites for information and products related to Salvo and its use:

- http://www.circuitcellar.com, "The magazine for Computer Applications," – lots of information on computer and embedded computer programming
- http://www.embedded.com – Home of *Embedded Systems Programming* magazine
- http://www.gnu.org – The Free Software Foundations GNU[159] project web server
- http://www.htsoft.com – HI-TECH Software LLC, home of the PICC, PICC Lite, PICC-18 and V8C compilers.
- http://www.metrowerks.com – Metrowerks Corporation, home of the CodeWarrior compiler and integrated development environment
- http://www.microchip.com – Microchip Corporation, supplier of PIC microcontrollers
- http://www.mixsoftware.com – Mix Software, Inc., home of the Power C compiler
- http://www.redhat.com – Provider of a well-known Linux distribution, and also home of the Cygwin[160] project.
- http://www.vautomation.com – VAutomation, Inc., home of the V8-µRISC™ synthesizeable 8-bit core

---

159 GNU is a recursive acronym for ``GNU's Not Unix"; it is pronounced "guh-NEW".

160 Search site for "Cygwin".

Appendix C • File and Program Descriptions

Overview

Each Salvo distribution contains a variety of tutorial, demo, test and other programs, as well as a multitude of other files. Most are intended for use on a particular target, although some – e.g. the Salvo source (`*.c` and `*.h`) files – are often universal.

Each distribution has an organized file hierarchy. Directories (i.e. folders) include subdirectories (i.e. subfolders), etc. Files that are higher up in a particular directory tree are more general, and those towards the bottom are more specific for a particular compiler and / or target.

If you have only one Salvo distribution, it will contain files for just your compiler and / or target processor. If you have multiple Salvo distributions, you should refer to Table 103 for the identifying name used for your particular compiler and target combination – the files you seek will be in those named subdirectories.[161]

Test Systems

A wide range of different test systems is used to verify Salvo's operation with different demo and test programs. The Salvo test systems are described in Table 103.

Name / Folder	Target	Compiler and IDE	Testbed
sysa	Microchip PIC16C77	HI-TECH PICC + Microchip MPLAB	Microchip PICDEM-2 demo board
sysb	Microchip PIC17C756	HI-TECH PICC + Microchip MPLAB	Proprietary data acquisition system

[161] E.g. the files and projects specific to the HI-TECH PICC-18 compiler and Microchip PIC18 PICmicro devices will reside in the `sysf` subdirectories.

sysc	x86 family	Mix Power C	generic Wintel platform
sysd	x86 family	Metrowerks CodeWarrior	generic Wintel platform
syse	Microchip PIC18C452	Microchip MPLAB-C18	Microchip PICDEM-2 demo board
sysf	Microchip PIC18C452	HI-TECH PICC-18 + Microchip MPLAB	Microchip PICDEM-2 demo board
sysg	Microchip PIC17C756A	HI-TECH PICC + Microchip MPLAB	SSDL/SCU PIC17 Prototoboard
sysh	Microchip PIC16F87X	HI-TECH PICC + Microchip MPLAB	Microchip MPLAB-ICD
sysi	Intel 8051 family	Keil C51 + µVision2	Cygnal C8051F005DK
sysj	Microchip PIC12C509	HI-TECH PICC + Microchip MPLAB	Salvo PIC12 Demo Board
sysk	Microchip PIC17C756	IAR PICmicro Embedded Workbench	SSDL/SCU PIC17 Prototoboard
sysl	VAutomation µV8-RISC	HI-TECH V8C	HI-TECH Simulator/V8 and VAutomation simV8
sysm	Intel 8051 family	HI-TECH HT51	Cygnal C8051F005DK
sysn	Intel 8051 family	TASKING 8051 C	Cygnal C8051F005DK
syso	Microchip PIC17C756	HI-TECH PICC	Pumpkin PIC17C75X Prototoboard
sysp	Microchip PIC18C452	IAR PIC18 C Compiler	Microchip PICDEM-2 demo board
sysq	Texas Instruments MSP430	IAR MSP430 + IAR Embedded Workbench	TI's MSP430 Simulator & MSP-FET430 Flash Emulation Tool

sysr	Texas Instruments MSP430	Archelon Quadravox AQ430 Tools	TI's MSP-FET430 Flash Emulation Tool
syss	Texas Instruments MSP430	ImageCraft ICC430 Development Tools	TI's MSP-FET430 Flash Emulation Tool
syst	Motorola M68HC11	ImageCraft ICC11 Development Tools	Motorola M68HC11 EVB

Table 103: Test System Names, Targets and Development Environments

In general, projects designed for a particular test system can be easily modified to work with other, similar target processors. For example, a `sysa` project could be recompiled for the Microchip PIC16F877 with minor changes, if any.

Projects

Nomenclature

All Salvo programs are built using projects. Usually the project type is the one native to the tool being used, e.g. Microchip MPLAB projects (`*.pjt`) or Keil µVision2 (`*.uV2`) projects.

Programs can be built using Salvo libraries or Salvo source code. Projects follow the naming convention shown below:

projectname`free.*`,
projectname`lite.*`: uses freeware libraries

projectname`lib.*`,
projectname`le.*`: uses standard libraries

projectname`.*`,
projectname`pro.*`: uses source code

projectname`ilib.*`,
projectname`prolib.*`: uses standard libraries with embedded debugging information

Note The `free`/`lib`/`ilib`/`(blank)` naming convention was used up to and including Salvo v3.0.5. As of v3.0.6, the `lite`/`le`/`pro`/`prolib` convention is used.

In many instances, a project may contain multiple project files, all using a single `salvocfg.h`.

Wherever possible, relative pathnames have been used in the project files to accommodate installations that do not use Salvo's default installation directory.

Note Programs built with freeware libraries are marked with a '†'.

Source Files

In most cases, we have avoided creating projects with identical or redundant source files. Each project generally contains the following:

project_dir`\main.c`
project_dir`\main.h`
project_dir`\`*other_source_files.c*
project_dir`\`*other_header_files.h*
project_dir`\`*test_system_dir*`\salvocfg.h`

`main.c` contains the source for the program. There may be additional source files in the project directory and / or in its test system subdirectories. `main.h` contains compiler- and target-specific symbols (if required). `salvocfg.h` contains the project-specific Salvo configuration.

Additionally, where several projects are grouped together (e.g. the tutorial projects `salvo\tut\tu1-tu6`), files that are common to all of the projects are located in the *first* project, even if they are not used by the first project in the group. Files that are common to a particular test system will be found in the associated folder (e.g. `salvo\tut\tu1\sysi`).

SYS Predefined Symbols

Preprocessor symbols in the form `SYSA`, `SYSB`, … (see Table 103) are used liberally within projects to control the conditional compilation of a project's header files. This is why a header file may con-

tain defined symbols that are unrelated to the compiler and/or target of your Salvo distribution.

File Types

The files found in Salvo's directories are summarized below. A description of the file, the file type (text, binary or executable) and the applications that use the file are listed for each file extension.

Note Some extensions are used by more than one program.

***.(no extension)**	Absolute object file	bin	Keil C51 linker
***.$$$**	Editor backup file	text	Microchip MPLAB
***.a**	Library (archive) file	bin	gcc compiler
***.asm**	Assembly language source file	text	editors, assemblers & compilers
***.bat**	MS-DOS batch file	text	DOS & Windows
***.c**	C language source file	text	editors & compilers
***.cod**	ByteCraft .COD file	bin	HI-TECH PICC
***.d43**	Debugging file	bin	IAR Embedded Workbench – MSP430
***.dbg**	Debugging file	text	ImageCraft ICC
***.dp2**	Dependency file	text	ImageCraft ICC
***.dtp**	Desktop layout file	bin	IAR Embedded Workbench
***.err**	Error file	text	various
***.exe**	Executable program	exe	DOS & Win-

			dows
***.h**	C-language header file	text	editors & compilers
***.hex**	Hex file suitable for download into emulator or device programmer	text	assemblers, compilers and linkers
***.inf**	Information file	text	Windows
***.ini**	Information file	text	IAR C-SPY debugger
***.lib**	Library file	bin	assemblers, compilers and linkers
***.lis**	Listing file	text	ImageCraft ICC, Mix Power C
***.lk**	Linker command file	text	ImageCraft ICC
***.lnp**	Linker input file to pass command line	text	Keil µVision2
***.lst**	Listing file	text	various compilers
***.M51**	Map file	text	Keil C51 toolset
***.mak**	Makefile	text	ImageCraft ICC
***.map**	Map file	text	various compilers
***.mcp**	Project file	bin	Metrowerks CodeWarrior
***.mix**	Object file	bin	Mix Power C
***.mp**	Map file	text	ImageCraft ICC
***.obj**	Object file	bin	HI-TECH PICC
***.Opt**	Local project option settings	text	Keil µVision2

***.pdf**	Portable document file	bin	Adobe Acrobat
***.pjt**	Project file	text	Microchip MPLAB
***.plg**	Protocol file that summarizes the last build process	text	Keil µVision2
***.pre**	C preprocessor output file	text	HI-TECH PICC
***.prj**	Project file	text	HI-TECH PICC, IAR Embedded Workbench, ImageCraft ICC
***.qin**	Information file	bin	Quadravox AQ430 Development Tools
***.qpj**	Project file	bin	Quadravox AQ430 Development Tools
***.r43**	object or library file	bin	IAR Embedded Workbench – MSP430
***.rlf**	Intermediate file	bin	HI-TECH PICC
***.rxc**	Project-related file	bin	Quadravox AQ430 Development Tools
***.s**	Assembly language source file	text	ImageCraft ICC
***.s43**	Assembly language source file	text	IAR Embedded Workbench – MSP430
***.sdb**	Symbolic debugging file	text	HI-TECH PICC
***.src**	Source file list	text	ImageCraft ICC
***.sym**	Symbol file	text	HI-TECH PICC

***.trc**	Trace file	text	Quadravox AQ430 Development Tools
***.txt**	Text file	text	editors
***.wat**	Watch window file	text	Microchip MPLAB
***.Uv2**	Project File	text	Keil µVision2

Included Projects and Programs

Demonstration Programs

demo\d1\sysa|e|f|t

Dual-mode program with 8 concurrent tasks and 5 events to demonstrate real-time, event-based multitasking. Designed for a mid-range Microchip PIC16C67/77 or similar PICmicro running at 4MHz on a Microchip PICDEM-2 demonstration board (i.e. Test System A). In mode 1 (delays), 8 tasks run with random delays, and the LEDs form a bargraph of the number of currently eligible tasks. In mode 2 (events), 5 of the tasks wait for semaphores signaled randomly by another task, and LEDs flash when each task runs.

In both modes, a "kernel dump" to an attached terminal (RS-232 at 9600, N, 8, 1) is available. It takes a "snapshot" and displays the statuses of the tasks and events, as well as various other run-time parameters.

Of Interest A single function is used for 6 of the 8 tasks, with different actions based on the taskID of the current task. Salvo uses only a small portion of the memory available, and performs over 3,000 context switches / second.

demo\d2\sysa|f|h

Similar to D1, but runs on a PIC16C64, which has less memory than D1's processor and no hardware USART. Implements RS-232

transmission via a software USART at 600 baud using 4MHz clock.

Of Interest Software USART will only work if interrupts are never disabled for more than 1/2 of a bit time. In D2, up to 8 tasks can run concurrently without violating this restriction.

demo\d3\sysa|j †

PWM fan speed controller with local and remote interfaces, all running on a baseline Microchip PIC12C509A PICmicro with only 1K of ROM and 41 bytes of RAM.

Of Interest `OSTimer()` used on interrupt-less target, fan speed and beeper controlled via pulsetrains with period resolution of a system tick, and three-wire software interface to latching serial shift register.

Please see *AN-6 Designing a Low-Cost Multifunction PIC12C509A-based Remote Fan Controller with Salvo* for more information.

demo\d4\sysa|e|f|h †

Four tasks with different priorities are used to:

- blink a single LED continuously at 1Hz
- count down a timer and display it when a key is pressed
- shift an LED is continuously across seven LEDs at a rate controlled by a potentiometer, and
- sample potentiometer position when the system is idling

Of Interest No matter what the lower-priority tasks are doing, the highest-priority task's timing is unaffected. Also, a single `main.c` is used to for three different target processor and target system combinations.

Example Programs

ex\ex1\sysa|e|f|h|p|q|r|s|t

Simple program for use with freeware libraries and *AN-1 Using Salvo Freeware Libraries with the HI-TECH PICC Compiler*.

Of Interest Prescalar for `OSTimer()` is done explicitly, since freeware libraries do not support Salvo's timer prescalar (`OSTIMER_PRESCALAR` is set to 0).

ex\ex2\sysa

Same as `example\ex1\sysa`, but adds `mem.c` as a project node and uses a different `salvocfg.h` to reduce Salvo's RAM utilization to the bare minimum.

Of Interest Salvo's RAM requirements are reduced substantially via this method.

Templates

Templates are small, self-contained programs that illustrate how to use certain Salvo services. They're useful for cut-and-pasting into your own applications.

tplt\te1

Three tasks running at same priority.

Of Interest In order for separate tasks to all run using only the `OS_Yield()` context switcher, they must all have the same priority.

Test Programs

test\t1\sysa|b|c|d

Salvo application that runs 8 tasks of equal priority. Used to measure the ROM and RAM requirements for simple multitasking.

Of Interest `t1` calls the three Salvo services are required for multitasking: `OSInit()`, `OSCreateTask()` and `OSSched()`. The target processor, compiler used and number of events are all specified in `salvocfg.h`. All other configuration options are left at their default values.

test\t2\sysa|b|c|d

Salvo application that runs 8 tasks of equal priority, each of which repeatedly delays itself for 1 system tick. Used to measure the ROM and RAM requirements for simple multitasking with delays. Builds on `t1`.

Of Interest `t2` adds a call to `OSTimer()` in order to support delay services. 8-bit delays are specified via `OSBYTES_OF_DELAYS` in `salvocfg.h`. `Qins.c` and `timer.c` are included in order to minimize the size of the interrupt context-save and -restore code.

test\t3\sysa|b|c|d

Salvo application that runs 8 tasks of equal priority and uses 6 events. Used to measure the ROM and RAM requirements for simple multitasking with events. Builds on `t1`.

Of Interest `t3` calls the three Salvo services which are necessary for using semaphores: `OSCreateSem()`, `OSSignalSem()`, `OS_WaitSem()`. A single task can signal multiple events (`TaskSignalSems()`), and can also wait on multiple events (`TaskWaitSems()`).

test\t4\sysa|b|c

Salvo application that runs 8 tasks of equal priority, uses 6 events and delays some tasks for 1 system tick. Used to measure the ROM

and RAM requirements for simple multitasking with events and delays. Combines `t2` and `t3`.

test\t5\sysa|b|c

Identical to `t4`, but supports timeouts, too

Of Interest `t5`'s `main.c` differs from `t4`'s only in the calls to `OS_WaitSem()`, which require a timeout parameter. Timeout support is enabled via `OSENABLE_TIMEOUTS` in `salvocfg.h`. Timeout support requires larger tcbs, therefore some versions use bank specifiers in `salvocfg.h`.

test\t6\sysa|b|c|d

Salvo application that runs just the idle function hook and counts context switches. Used to measure the best-case context switching rate.

Of Interest Idle function hook is enabled via `OSENABLE_IDLING_HOOK` in `salvocfg.h`.

test\t7\sysa|b|c|d

Salvo application that runs 5 tasks of equal priority. Used to measure the context-switching rate for multiple tasks at the same priority, which is dependent on the queueing algorithm and number of tasks.

Of Interest Round-robin scheduling is achieved by assigning all the tasks the same priority.

test\t8\sysa|b|c|d

Salvo application that runs 5 tasks of different priorities. Used to measure the context-switching rate for multiple tasks at the same priority, which is dependent on the queueing algorithm and number of tasks.

Of Interest Non-circular queueing algorithm (default) inserts from head of queue. Since only the highest-priority task is running

(all others remain eligible), queueing times are short, and therefore context-switching rate is high.

test\t9\sysa|b|c|d

Obsolete (used circular queues).

test\t10\sysa|b|c|d

Obsolete (used circular queues).

test\t11\sysa

Test program to obtain t_InsPrioQ for test configurations I & III.

test\t12\sysa

Test program to obtain t_InsPrioQ for test configurations II & IV.

test\t13\sysa

Test program to obtain t_InsPrioQ for test configuration V.

test\t14\sysa

Test program to obtain execution speeds for:

- `OS_Destroy()`,
- `OS_Prio()`,
- `OS_Stop()`,
- `OS_WaitMsg()`,
- `OS_WaitSem()`,
- `OS_Yield()`,
- `OSCreateMsg()`,
- `OSCreateSem()`,
- `OSCreateTask()`,
- `OSInit()`,
- `OSSched()`,
- `OSSignalMsg()`,
- `OSSignalSem()` and
- `OSStartTask()`

Configuration III is used because events are supported.

test\t15\sysa

Test program to obtain execution speeds for:

- `OS_Delay()` and
- `OSTimer()`

Configuration II is used because delays are supported.

test\t16\sysa

Test program to verify proper operation of explicit task-control services like `OSStartTask()` and `OSStopTask()`.

test\t17\sysa

Test program to obtain t_DelPrioQ for test configurations I & III.

test\t18\sysa

Test program to obtain t_DelPrioQ for test configurations II & IV.

test\t19\sysa

Test program to obtain t_InsDelayQ for test configurations II & IV, with 8-bit delays.

test\t20\sysa

Test program to obtain t_InsDelayQ for test configurations II & IV, with 16-bit delays.

test\t21\sysa

Test program to obtain t_InsDelayQ for test configurations II & IV, with 8-bit delays, using `OSSPEEDUP_QUEUEING`.

test\t22\sysa

Test program to obtain t_InsDelayQ for test configurations II & IV, with 16-bit delays, using `OSSPEEDUP_QUEUEING`.

test\t23\sysa

Test program to obtain t_InsDelayQ for test configuration V, with 8-bit delays.

test\t24\sysa

Test program to obtain t_InsDelayQ for test configuration V, with 16-bit delays.

test\t25\sysa

Test program to obtain t_InsDelayQ for test configuration V, with 8-bit delays, using `OSSPEEDUP_QUEUEING`.

test\t26\sysa

Test program to obtain t_InsDelayQ for test configuration V, with 16-bit delays, using `OSSPEEDUP_QUEUEING`.

test\t27\sysa

Test program to obtain t_DelDelayQ for test configurations II & IV, with 8-bit delays.

test\t28\sysa

Test program to obtain t_DelDelayQ for test configurations II & IV, with 16-bit delays.

test\t29\sysa

Test program to obtain t_DelDelayQ for test configuration V, with 8-bit delays.

test\t30\sysa

Test program to obtain t_DelDelayQ for test configuration V, with 16-bit delays.

test\t31\sysa

Test program to verify proper operation of message queues.

test\t32\sysa

Test program to verify proper operation of Salvo signaling services called from both mainline code and interrupts via `OSFROM_ANYWHERE` configuration option.

test\t33\sysa

Test program to verify proper array mode operation.

test\t34\syse|f

Test program to verify PIC18C PICmicro ports.

test\t35\syso

Test program to verify the basic hardware functionality of Pumpkin's PIC17C75X Protoboard.

Of Interest Simple software SPI implementation.

test\t36\sysa

Test program to verify simple task switching among tasks with equal priorities.

test\t37\sysf

Test program to verify PICC-18's `indir_func` (call by pointer) library function.

test\t38

Test program to verify Salvo functionality on a Microchip 12-bit PICmicro (e.g. PIC16C57).

Of Interest The 12-bit PICmicro MCUs do not have interrupts. Therefore to use Salvo's time services, `OSTimer()` must be called from mainline code. By monitoring the free-running Timer0 and calling `OSTimer()` each time it rolls over, a reliable system tick rate is achieved:

```
tmpTMR0 = TMR0;
if ( tmpTMR0 < oldTMR0 )
    OSTimer();
oldTMR0 = tmpTMR0;
```

Also, HI-TECH PICC circumvents the limitations of a 2-level-deep call…return stack by managing function calls and returns via a jump table.

test\t39

Unused.

test\t40-t47\sysa|e|f|l|p|q|r|s|t

Test programs for functional testing and Salvo certification.

Of Interest This series of test programs use the target processor's output ports to indicate various activities of the test program. By connecting these ports to a logic analyzer, proper operation of the Salvo test program can be verified. These programs are used to certify new compilers and/or target processors.

Test programs `t40-t47` are all based on the same source code. Compilation is controlled through preprocessor symbols `TEST_XYZ`, listed in Table 104. The source files used for each program are listed for reference.

test program	source files	defined symbol(s)
t40	main.c, salvo\init.c, salvo\mem.c	(none)
t41	+salvo\qins.c, +salvo\sched.c, +salvo\util.c	TEST_SCHEDULER
t42	+salvo\inittask.c	TEST_SCHEDULER, TEST_YIELDING_TASKS
t43	+salvo\event.c, +salvo\sem.c	TEST_SCHEDULER, TEST_WAITING_TASKS
t44	+isr.c, -salvo\event.c, -salvo\inittask.c, -salvo\sem.c	TEST_INTERRUPTS, TEST_SCHEDULER
t45	+salvo\timer.c	TEST_INTERRUPTS, TEST_SCHEDULER, TEST_TIMER
t46	+salvo\delay.c, +salvo\inittask.c	TEST_DELAYED_TASKS, TEST_INTERRUPTS, TEST_SCHEDULER, TEST_TIMER
t47	+salvo\event.c, +salvo\sem.c	TEST_INTERRUPTS, TEST_SCHEDULER, TEST_TIMER, TEST_WAITING_TASKS

Table 104: Configurations for Test Programs t40-t47

Tutorial Programs

The tutorial programs are described in-depth in *Chapter 4 • Tutorial*. Each tutorial can be built using the freeware libraries, the standard libraries or the source code.

tut\tu1\sysa|e|f|h|i|l|q|r|s|t †

A minimal Salvo application comprised of a call to `OSInit()` followed by `OSSched()` called from within an infinite loop.

tut\tu2\sysa|e|f|h|i|l|q|r|s|t †

A multitasking Salvo application with two tasks. Introduces `OSCreateTask()` and `OS_Yield()` for task management and context switching.

Of Interest Both tasks run at the same priority in order to round-robin.

tut\tu3\sysa|e|f|h|i|l|q|r|s|t †

Multitasking with two non-trivial tasks.

Of Interest Two separate processes (a counter incrementing and writes to an output port) appear to occur simultaneously when viewed by the user. Also, tasks have a clearly-defined initialization portion that runs only once. The tasks are tightly-coupled.

tut\tu4\sysa|e|f|h|i|l|q|r|s|t †

Multitasking with an event. Introduces `OSCreateSem()`, `OSSignalSem()` and `OS_WaitSem()` for event (semaphore) management.

Of Interest Output task waits until free-running counter task signals the semaphore. Then it updates the output port and resumes waiting. The tasks are loosely coupled.

tut\tu5\sysa|e|f|h|i|l|q|r|s|t †

Multitasking with a delay. Introduces `OS_Delay()` and `OSTimer()` for time-based services.

Of Interest `OSTimer()` is tied to a periodic interrupt, and delay is specified as a number of system ticks. The tasks are loosely coupled.

tut\tu6\sysa|e|f|h|i|l|q|r|s|t †

Signaling from multiple tasks. Introduces `OSCreateMsg()`, `OSSignalMsg()` and `OS_WaitMsg()`. A message can be signaled from one of two tasks, and is waited on by a third.

Of Interest A single, waiting task will react differently upon receipt of a message depending on the message's contents. The tasks are loosely coupled.

Also, extra configuration options in `salvocfg.h` can be used to minimize the RAM requirements of the projects using the freeware and standard libraries.

Library Files

lib*.*

Precompiled Salvo freeware and standard libraries for a variety of compilers and targets. See *Chapter 8 • Libraries* for more information.

Third-Party Files

free\links*.*

Links to various URLs for free programs related to using Salvo.

Index

µ

µC/OS ..... *See* MicroC/OS-II

A

additional documentation
  application notes 517
  assembly guides 517
assembly language xxxi
  portability 25

C

C compiler 500
C programming language 518
  portability 26
compiler
  recompile (re-make) 499
  search paths 496, 497
complex expressions in Power C 511
complexity
  application 11, 89
  managing 212
  scheduler 19
  size vs. speed 183
configuration options
  OSBIG_MESSAGE_POINTERS .. 110, 112, 131, 132, 387, 409, 414, 506
  OSBIG_SEMAPHORES 110, 112, 131, 132, 294, 348
  OSBYTES_OF_COUNTS 110, 112, 133, 170, 328, 387
  OSBYTES_OF_DELAYS ... 87, 89, 96, 110, 112, 135, 136, 143, 194, 227, 229, 230, 268, 312, 328, 356, 387, 388, 462, 533
  OSBYTES_OF_EVENT_FLAGS . 110, 112, 119, 134, 153, 272, 288
  OSBYTES_OF_TICKS . 110, 112, 136, 178, 194, 227, 231, 232, 260, 308, 310, 338, 340, 356
  OSCALL_OSCREATEEVENT 111, 113, 137, 138, 139, 140, 141, 286, 288, 290, 292, 294
  OSCALL_OSGETPRIOTASK 140

OSCALL_OSGETSTATETASK 140
OSCALL_OSMSGCOUNT 111
OSCALL_OSMSGQCOUNT 140, 314
OSCALL_OSMSGQEMPTY 111, 140, 316
OSCALL_OSRETURNEVENT 111, 113, 137, 141, 154, 155, 318, 320, 322, 324, 326, 358, 360, 362, 364
OSCALL_OSSIGNALEVENT 111, 113, 137, 141, 284, 333, 342, 344, 346, 348
OSCALL_OSSTARTTASK 111, 113, 141
OSCLEAR_GLOBALS 110, 112, 142, 312, 467, 479
OSCLEAR_UNUSED_POINTERS 111, 113, 143, 330
OSCLEAR_WATCHDOG_TIMER() 111, 144, 227, 499, 500
OSCOMBINE_EVENT_SERVICES 111, 112, 145, 255, 284, 286, 288, 290, 292, 294, 332, 342, 344, 346, 348
OSCOMPILER .. 87, 96, 110, 114, 116, 125, 131, 142, 203, 225, 226, 378, 380, 401, 408, 413, 417, 421, 425, 428, 431, 434, 437, 440, 443, 446, 449, 484, 485, 491, 498
OSCTXSW_METHOD 111, 113, 114, 115, 147, 190
OSDISABLE_ERROR_CHECKING 111, 148, 152, 356
OSDISABLE_FAST_SCHEDULING 111, 113, 149
OSDISABLE_TASK_PRIORITIES..... 150, 266, 296, 300, 302, 334, 336
OSENABLE_BINARY_SEMAPHORES..... 110, 112, 118, 151, 153, 161, 162, 164, 270, 286, 318, 342, 358
OSENABLE_BOUNDS_CHECKING..... 152, 192
OSENABLE_EVENT_FLAGS.. xxxiii, 112, 119, 134, 151, 153, 161, 162, 164, 272, 284, 288, 320, 332
OSENABLE_EVENT_READING 110, 112, 154, 155, 318, 320, 322, 324, 326, 358, 360, 362, 364
OSENABLE_EVENT_TRYING 110, 112, 154, 155
OSENABLE_FAST_SIGNALING 110, 112, 156
OSENABLE_IDLE_COUNTER..... 110, 112, 157
OSENABLE_IDLING_HOOK 110, 111, 112, 157, 158, 234, 384, 484, 534
OSENABLE_INTERRUPT_HOOKS..... 111, 159, 370, 382
OSENABLE_MESSAGE_QUEUES 110, 112, 118, 124, 151, 153, 161, 162, 164, 278, 292, 314, 316, 324, 346, 362
OSENABLE_MESSAGES..... 96, 110, 112, 118, 151, 153, 161, 162, 164, 276, 322, 344, 360
OSENABLE_SCHEDULER_HOOK..... 111, 163
OSENABLE_SEMAPHORES 110, 112, 118, 151, 153, 161, 162, 164, 226, 280, 294, 326, 348, 364
OSENABLE_STACK_CHECKING..... 111, 112, 142, 165, 170, 175, 220, 258, 260, 262, 266, 268, 270, 272, 276, 278, 280, 284, 286, 288, 290, 292, 294, 296, 298, 300, 302, 304, 306,

308, 310, 312, 328, 330, 332, 334, 336, 338, 340, 342, 344, 346, 348, 350, 352, 356
OSENABLE_TIMEOUTS 110, 143, 169, 176, 231, 270, 272, 276, 280, 459, 534
OSEVENT_FLAGS..... 118, 119, 128, 153, 380, 402
OSEVENTS . 88, 89, 96, 110, 112, 118, 126, 128, 151, 153, 161, 162, 164, 175, 192, 203, 246, 270, 272, 276, 278, 280, 284, 286, 288, 290, 292, 293, 294, 312, 318, 320, 322, 324, 326, 332, 342, 344, 346, 348, 358, 360, 362, 364, 380, 402, 467
OSGATHER_STATISTICS .. 111, 112, 133, 157, 165, 170, 176, 179, 226
OSINTERRUPT_LEVEL..... 111, 171
OSLIBRARY_CONFIG 111, 113, 115, 120, 121, 122, 123, 127, 128, 203, 204, 206, 401, 402, 404, 409, 414, 418, 422, 426, 429, 432, 435, 437, 441, 443, 446, 450, 498
OSLIBRARY_GLOBALS 121
OSLIBRARY_TYPE..... 111, 113, 115, 120, 121, 122, 123, 127, 128, 203, 204, 206, 401, 402, 403, 409, 414, 418, 422, 426, 429, 432, 435, 437, 441, 443, 446, 449, 498
OSLIBRARY_VARIANT..... 111, 113, 115, 120, 121, 122, 123, 127, 128, 203, 204, 206, 401, 402, 405, 409, 414, 422, 426, 429, 444, 446, 450, 498
OSLOC_ALL..... 111, 113, 172, 174, 204, 222
OSLOC_COUNT.... 111, 113, 172, 174, 175, 176, 177, 178, 389
OSLOC_CTCB..... 111, 113, 172, 175, 389
OSLOC_DEPTH..... 111, 113, 172, 175, 389
OSLOC_ECB..... 89, 111, 113, 172, 175, 202, 388, 389
OSLOC_EFCB 175
OSLOC_ERR..... 111, 113, 172, 176, 389
OSLOC_GLSTAT 176, 389
OSLOC_LOGMSG..... 111, 113, 172, 176, 389
OSLOC_MQCB..... 111, 113, 124, 172, 176, 293, 388, 389
OSLOC_MSGQ..... 111, 113, 124, 172, 177, 293, 388, 389
OSLOC_PS..... 111, 113, 172, 177, 389
OSLOC_SIGQ 111, 113, 172, 178, 389
OSLOC_TCB..... 111, 113, 166, 172, 177, 202, 388, 389
OSLOC_TICK..... 111, 113, 172, 178, 389
OSLOG_MESSAGES 111, 112, 114, 176, 179, 180, 181
OSLOGGING 111, 112, 114, 170, 179, 180, 181, 220, 258, 260, 268, 270, 272, 276, 278, 280, 284, 286, 288, 290, 292, 294, 296, 312, 330, 332, 342, 344, 346, 348, 350
OSMESSAGE_QUEUES..... 110, 118, 124, 128, 162, 177, 246, 292, 293, 380, 402
OSMPLAB_C18_LOC_ALL_NEAR 182
OSOPTIMIZE_FOR_SPEED..... 110, 112, 183, 186, 330
OSPIC16_GIE_BUG..... 111, 520

OSPIC18_INTERRUPT_MASK..... 111, 184, 185
OSPRESERVE_INTERRUPT_MASK..... 111, 186, 430
OSRPT_HIDE_INVALID_POINTERS. 111, 113, 187, 188, 189
OSRPT_SHOW_ONLY_ACTIVE..... 111, 113, 187, 188, 189
OSRPT_SHOW_TOTAL_DELAY..... 111, 113, 187, 188, 189
OSRTNADDR_OFFSET..... 111, 113, 147, 190
OSSCHED_RETURN_LABEL()..... 191
OSSET_LIMITS..... 152, 192, 403, 450
OSSPEEDUP_QUEUEING .. 110, 112, 193, 463, 471, 473, 477, 478, 536, 537
OSTARGET. 87, 88, 96, 110, 114, 116, 125, 203, 225, 226, 401, 408, 413, 417, 421, 425, 428, 431, 434, 437, 440, 443, 446, 449, 484, 485, 498
OSTASKS..... 65, 88, 96, 97, 110, 112, 118, 126, 128, 204, 226, 234, 238, 239, 312, 380, 402, 467, 484, 498
OSTIMER_PRESCALAR89, 110, 111, 113, 135, 136, 194, 228, 229, 230, 231, 232, 356, 532
OSUSE_EVENT_TYPES 111, 113, 197, 284, 286, 288, 290, 292, 294, 328, 332, 342, 344, 346, 348
OSUSE_INLINE_OSSCHED 111, 113, 198, 199, 220, 330
OSUSE_INLINE_OSTIMER. 111, 113, 198, 200, 220, 251, 356
OSUSE_INSELIG_MACRO..... 198, 201, 253
OSUSE_LIBRARY 111, 113, 120, 121, 122, 123, 127, 128, 203, 204, 206, 393, 400, 401, 402, 409, 414, 418, 422, 426, 429, 432, 435, 437, 441, 443, 446, 449, 498
OSUSE_MEMSET 111, 113, 202
other
MAKE_WITH_FREE_LIB 203, 204, 206
MAKE_WITH_STD_LIB 203, 204
SYSA … SYSZ..... 205, 206
SYSA…SYSZ..... 205
USE_INTERRUPTS..... 207
conflicts
deadlock..... 38
priority inversion..... 39, 520
context switch 12
critical section 18
CTask 519

D

debugging..... 495
breakpoints..... 499
delay..... *See* task
demo version..... *See* installation:demo version
demonstration programs

descriptions 530

E

event flags 13, 242
events 13
response time 20
example programs
descriptions 532
examples
how to
allow access to a shared resource 287
ascertain which event flag bit(s) are set 321
avoid overfilling a message queue 315, 317
build a library without command-line tools 497
change a task's priority on-the-fly 267
change a task's priority from another task 337
check a message before signaling 323
clear an event flag after successfully waiting it 285
context-switch outside a task's infinite loop 341
context-switch unconditionally 283
count interrupts 383
create a task 297
create an 8-bit event flag 289
define a null function 497
destroy a task 299
detect a timeout 375
directly read the system timer 309
directly write the system timer 339
dispatch most eligible task 331
display Salvo status 329
generate a single pulse 343
get current task's taskID 311
get current task's timestamp 311
get system ticks 309
initialize a ring buffer 295
initialize an LCD controller without delay loops 259, 261
initialize Salvo 313
manage access to a shared resource 345
obtain a message from within an ISR 361
obtain the current task's priority 301, 303, 305, 307
pass a keypress in a message 290
pass raw data using messages 249
phase-shift a task 355
preserve a task's timestamp 341
print the version number 377

process a buffer only when it is non-empty 281
protect a critical section of code 371
protect a service called from foreground and background.. 373
protect Salvo variables against power-on reset..... 222, 232
read a binary semaphore's value 319
read a semaphore's value..... 327
replace one task with another using only one taskID 265
reset a binary semaphore by reading it 359
reuse a taskID..... 263
rotate a message queue's contents..... 363
run a task for a one-time event..... 271, 273
run a task only once 269
run an idling function alongside Salvo 367
run incompatible code alongside Salvo 367
run OSTimer() from an interrupt 357
run OSTimer() from mainline code 531, 539
set a task's timestamp when it starts..... 341
set system ticks 339
start a task 351
stop a task..... 353
test a message in a message queue..... 324
toggle a port bit when idling..... 385
use the persistent type qualifier..... 174
vary a task's priority based on global variable..... 335
wait for a keypress in a message..... 277
wake another task..... 349
wake two tasks simultaneously..... 333
of
different task structures..... 21
multiple delays in a task..... 4
non-reentrant function behavior..... 15
salvocfg.h for use with freeware library 128
specifying register bank 0 in Hi-Tech PICC..... 172, 174
using #define to improve legibility..... 68, 72, 76, 88

F

foreground / background systems 11, 14–15
freeware version of Salvo .. xxxii, 57, 115, 211, 399, 432, 435, 438, 441

H

Harbison, Samuel P..... 518

I

idle task 112, 234
  priority 237
idling 13
installation
  avoiding long pathnames 55
  demo version 57
  directories
    demos 61, 226, 329
    include files 83, 86, 97, 114, 225, 391, 392, 393, 403, 485, 496, 510
    libraries 85, 98, 99, 403
    source files ... 83, 100, 101, 102, 391, 392, 393, 401, 453, 496, 507
    test programs 226, 455, 493, 508
    tutorials 61, 63, 66, 68, 72, 74, 76, 81, 86, 91, 92, 94, 96, 100, 104, 105, 106, 526
  Internet 56
  non-Wintel platforms 57
  on a network 57
  restoring source code 55
  serial number 51, 54
interrupt service routine (ISR) 12, 14
  calling Salvo services from 250
  compiler-generated context saving 228
  OSTimer() 74, 228, 232, 356
  priorities 240
  requirements 17
  response times 20
  restrictions on calling Salvo services 235
  salvocfg.h 245
  stack depth 220
  static variables 236
  use in
    foreground / background systems 14
    intertask communications 13
interrupt_level pragma (HI-TECH PICC compiler) .. 139, 410, 412, 414, 416, 444, 508
interrupts .....12, 14–15, 249–51. *See* interrupt service routine (ISR)
  avoiding problems with reentrancy 16
  calling Salvo services from 246
  controlling 485–86
  debugging 499, 500
  dis- and enabling in scheduler 467
  disabled 463
  effect on performance 218

in cooperative multitasking 20–21
in preemptive multitasking 18–20
including source files for optimum performance 90
interrupt level #pragma 508
latency 18, 229
periodic 25, 74, 228
polling 216
recovery time 20
response time 20
Salvo configuration options 111
software USART 531
using OSTimer() without 232, 531, 539
intertask communication 13

K

Kalinsky, David 520
kernel 12, 16
Kernighan, Brian W. 518

L

Labrosse, Jean J. 518, 519
LaVerne, David 519
libraries
freeware libraries 84, 97, 100, 105, 129, 412, 416, 497, 517, 532
linking
errors 98, 102
Linux 59
Linux / Unix . xxxi, 59, 408, 413, 417, 425, 428, 451, 452, 501, 521
Cygwin Unix environment for Windows 448, 452, 453, 501, 521

M

make utility 81, 85, 90
map file 104
memory requirements 103
message queues 13, 37
messages 13, 35
receiving 36
signaling 36
use in place of binary semaphores 37
MicroC/OS-II 518
multcall.h 116, 137
multitasking 16, 21

event-driven 28
mutexes 520
mutual exclusion 16

O

operating system (OS) 14

P

persistent type qualifier 222
PIC17C75X Protoboard 524, 538
pointer 35
declaring multiple 390
dereferencing 35
null 36
runtime bounds checking 152
predefined constants 64, 114, 147, 190, 238
OSCALL_OSCREATEEVENT
OSFROM_ANYWHERE 114, 137, 138, 139, 372, 538
OSFROM_BACKGROUND 137, 138
OSFROM_FOREGROUND 137, 138
OSCALL_OSXYZ
OSFROM_ANYWHERE 114, 137, 538
OSFROM_BACKGROUND 137
OSFROM_FOREGROUND 137
OSCOMPILER
OSAQ_430 ..... 114, 116
OSHT_8051C 114, 116
OSHT_PICC 87, 96, 114, 116, 203
OSHT_V8C 114, 116
OSIAR_ICC 114
OSKEIL_C51 ..... 114, 116
OSMIX_PC 114, 116
OSMPLAB_C18 ..... 111, 113, 114, 116, 173, 182
OSMW_CW 114, 116
OSCTXSW_METHOD
OSRTNADDR_IS_PARAM 147
OSRTNADDR_IS_VAR 114, 115, 147, 190
OSLOGGING
OSLOG_ALL 114, 180
OSLOG_ERRORS 114, 180
OSLOG_NONE 114, 180
OSLOG_WARNINGS 114, 180
OSStartTask()
OSDONT_START_TASK 64, 238, 264, 296, 351

OSTARGET
OSMSP430 114
OSPIC12 ..... 114, 125
OSPIC16 ..... 87, 96, 111, 114, 125, 203, 209, 520
OSPIC17 ..... 114, 125
OSPIC18 ..... 114, 125, 184
OSX86..... 114, 125
OSVERSION 376
preemption 12
printf() 15, 180, 328, 500
program counter 16, 17

R

RAM
reducing freeware library requirements 224
real-time operating system (RTOS) 14
reentrancy..... 15
resources
managing via semaphores 33
Ritchie, Dennis M. 518
round-robin 22, 237

S

salvo.h 3, 61, 62, 63, 66, 68, 72, 74, 76, 82, 83, 96, 97, 100, 116, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 131, 132, 133, 134, 135, 136, 137, 142, 143, 144, 145, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 161, 162, 163, 164, 165, 166, 169, 170, 171, 172, 174, 179, 180, 182, 183, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 200, 201, 202, 208, 226, 258, 260, 262, 264, 266, 268, 270, 272, 276, 278, 280, 282, 366, 368, 370, 374, 376, 378, 380, 382, 391, 484, 485, 496
including 82
locating..... 83, 97
salvocfg.h xxxiii, 82, 83, 84, 85, 86, 87, 88, 89, 95, 96, 97, 99, 110, 114, 116, 117, 120, 121, 122, 123, 127, 128, 144, 151, 153, 154, 155, 161, 162, 164, 168, 172, 174, 203, 204, 205, 206, 207, 209, 214, 215, 220, 224, 225, 226, 227, 231, 239, 245, 288, 293, 370, 393, 400, 401, 402, 403, 404, 405, 408, 409, 413, 414, 417, 418, 421, 422, 425, 426, 428, 429, 431, 432, 434, 435, 437, 440, 441, 443, 444, 446, 449, 450, 460, 461, 484, 485, 496, 497, 498, 499, 500, 506, 509, 526, 532, 533, 534, 542
changing..... 85
default 225, 393, 460, 461

default values 95
editing 114
identifying the compiler 87
including 82
leaving a configuration option undefined 89
locating 83, 86, 97, 114
specifying the number of events 88
specifying the number of tasks 88
using MAKE_WITH_XYZ_LIB for two sets of configuration options in one file 203
scheduling 12, 16, 24
semaphores 13, 29
shared resources 16
stack 12, 19
call ... return 5
general-purpose 5
hardware ..... *See* call ... return stack
overcoming limitations 252
role in reentrancy 16
saving context 17
Steele, Guy L., Jr. 518
superloop ..... 11, 14. *See* foreground / background systems
symbolic debugging 107
synchronization
conjunctive ..... *See* event flags
disjunctive ..... *See* event flags
system response 15
system timer ..... *See* timer

T

task 12
association with events 28
behavior
due to context switch 17
during interrupts 17–18
in cooperative multitasking 20–21
in preemptive multitasking 18–20
context 12, 17
delay 13, 24–26
in-line loop 25
maximum 25
using timer 26
preemption 12
priority 12
dynamic 22

importance thereof 217
static 22
priority-based execution 22
relationship to events 13
round-robin execution 22
running 12
state 13, 23–24
transition 23
structure 21–22
suspending and resuming 12
switch ..... *See* context switch
synchronization 31
templates
descriptions 532
test programs 457
descriptions 533–39
timeouts 13
breaking a deadlock with 38
timer 13
accuracy 26
resolution 26
system tick 25
system tick rate 25
using OSTimer() without interrupts 232
tools
HI-TECH Software
HPDPIC integrated development environment . 91, 92, 97, 98, 99, 100, 101, 103, 105, 408, 413, 502, 504, 505, 508
mouse problems 502
running in DOS window 502
running under Windows 2000 502
HPDV8 integrated development environment 508
PICC compiler . 8, 84, 86, 87, 89, 91, 95, 96, 97, 98, 103, 104, 105, 114, 116, 125, 127, 128, 129, 131, 137, 138, 139, 144, 171, 172, 173, 174, 191, 196, 203, 219, 222, 252, 328, 401, 408, 409, 410, 411, 412, 413, 414, 415, 416, 443, 444, 453, 454, 456, 483, 484, 486, 487, 488, 490, 497, 501, 502, 503, 504, 505, 506, 507, 508, 509, 515, 517, 521, 523, 524, 527, 528, 529, 530, 532, 538, 539
PICC-18 compiler .. 8, 116, 138, 171, 173, 413, 483, 484, 521, 523, 524, 538
IAR Systems
C-SPY Debugger 417, 432, 435, 437, 438, 441, 528
MSP430 C compiler 116, 437, 438, 439, 524
in-circuit debugger (ICD) 499
in-circuit emulator (ICE) 499

Keil
Cx51 Compiler..... 116, 172, 173, 196, 524, 527, 528
make utility 81, 85, 90, 501
Metrowerks
CodeWarrior C compiler 8, 116, 449, 450, 484, 486, 488, 490, 498, 512, 513, 521, 524, 528
Microchip
MPLAB integrated development environment 84, 86, 91, 105, 106, 107, 116, 128, 173, 182, 184, 204, 206, 390, 408, 413, 417, 421, 422, 423, 424, 484, 499, 503, 513, 517, 523, 524, 525, 527, 529, 530
MPLAB-C18 C compiler... 116, 173, 182, 184, 421, 422, 423, 424, 484, 524
MPLAB-ICD in-circuit debugger..... 499
MPLAB-ICE in-circuit emulator..... 499
PICMASTER in-circuit emulator 91, 106, 499
Microchip, Inc.
MPLAB-C18 compiler..... 173, 182
Mix Software
Power C compiler 116, 236, 456, 484, 486, 487, 489, 490, 510, 511, 512, 518, 521, 524, 528
Power C debugger..... 512
Quadravox
AQ430 Development Tools116, 431, 432, 434, 435, 436, 440, 441, 525, 529, 530
tutorial..... 61, 63, 66, 68, 72, 76, 80, 81, 86, 91, 92, 94, 95, 96, 97, 100, 104, 105, 106, 203, 215, 226, 518, 523, 526, 540, 541
program descriptions..... 540
typecasting 78, 245, 514, 515
types
predefined .....*See* variables:Salvo defined types

U

uninstaller..... 58
user macros
_OSLabel(). 3, 63, 64, 66, 68, 69, 72, 73, 76, 233, 234, 265, 378, 379, 488, 489, 490
OSECBP()..... 68, 72, 76, 89, 192, 221, 271, 273, 285, 287, 289, 291, 293, 295, 315, 317, 325, 333, 343, 345, 380, 381
OSEFCBP()..... 380
OSMQCBP()..... 292, 293, 380
OSTCBP() 4, 63, 65, 66, 68, 72, 76, 88, 167, 221, 234, 237, 238, 239, 261, 265, 267, 297, 303, 307, 331, 351, 369, 380, 381, 402
user services

events
OS_WaitBinSem() 118, 151, 242, 254, 270, 271, 286, 318, 342, 343, 353, 355, 358, 359, 374, 379, 464
OS_WaitEFlag() 115, 153, 272, 273, 274, 275, 284, 285, 288, 320, 321, 332, 333
OS_WaitMsg()77, 78, 118, 161, 197, 231, 241, 242, 245, 247, 259, 276, 277, 279, 290, 291, 322, 345, 360, 374, 375, 465, 515, 535, 541
OS_WaitMsgQ() 118, 124, 162, 278, 279, 293, 314, 316, 317, 324, 346, 362, 374, 465
OS_WaitSem()..... 69, 70, 71, 72, 73, 118, 164, 221, 230, 244, 280, 281, 294, 326, 348, 364, 374, 465, 489, 533, 534, 535, 541
OSClrEFlag() 141, 153, 273, 274, 275, 284, 285, 320, 333, 394
OSCreateBinSem() 118, 138, 139, 145, 151, 271, 286, 287, 318, 319, 342, 343, 358, 394, 466
OSCreateEFlag() 119, 153, 273, 274, 284, 288, 289, 320, 333, 394
OSCreateMsg() 77, 78, 100, 118, 161, 245, 277, 286, 290, 291, 322, 345, 360, 375, 394, 466, 535, 541
OSCreateMsgQ() 118, 124, 145, 162, 279, 292, 293, 314, 317, 324, 346, 362, 394, 466
OSCreateSem() 69, 70, 73, 118, 164, 221, 244, 248, 281, 294, 295, 326, 327, 348, 349, 353, 364, 365, 381, 394, 466, 533, 535, 541
OSMsgQCount() 140, 314
OSMsgQEmpty() 140, 316, 317, 394, 406
OSReadBinSem() 154, 271, 286, 318, 319, 342, 358, 394, 406
OSReadEFlag() .. 141, 154, 273, 284, 288, 320, 321, 333, 394, 406
OSReadMsg() 154, 277, 290, 322, 323, 345, 360, 394, 406
OSReadMsgQ().. 154, 279, 293, 314, 317, 324, 325, 346, 362, 394, 406
OSReadSem() 154, 281, 294, 326, 327, 348, 364, 394, 406
OSSetEFlag() 141, 153, 273, 274, 275, 332, 333
OSSignalBinSem().... xxxi, 118, 151, 156, 251, 270, 271, 286, 318, 342, 343, 352, 358, 372, 373, 395, 406, 468
OSSignalMsg() . xxxi, 76, 77, 78, 79, 100, 118, 141, 146, 161, 197, 220, 241, 245, 246, 247, 250, 277, 290, 291, 322, 344, 345, 346, 360, 387, 395, 406, 468, 515, 535, 541
OSSignalMsgQ()..... xxxi, 118, 124, 162, 279, 293, 314, 316, 317, 324, 346, 347, 362, 363, 395, 406, 468
OSSignalSem() ... xxxi, 69, 70, 71, 72, 73, 118, 164, 197, 221, 244, 250, 281, 294, 326, 348, 349, 364, 395, 404, 406, 468, 469, 533, 535, 541

OSTryBinSem() 155, 271, 286, 318, 342, 358, 359, 395
OSTryMsg() 155, 277, 290, 322, 345, 360, 361, 395
OSTryMsgQ() 155, 279, 293, 314, 317, 324, 346, 362, 363, 395
OSTrySem() 141, 155, 281, 294, 326, 348, 364, 365, 395
general
OSInit() 4, 62, 63, 66, 69, 73, 77, 100, 126, 142, 167, 198, 199, 222, 239, 244, 260, 261, 265, 297, 308, 312, 313, 330, 331, 338, 350, 394, 410, 415, 444, 460, 466, 467, 479, 485, 491, 498, 500, 533, 535, 540
OSSched() .. 4, 62, 63, 67, 69, 74, 77, 100, 126, 144, 157, 158, 163, 165, 167, 175, 191, 198, 235, 237, 239, 244, 245, 251, 261, 265, 297, 313, 330, 331, 351, 363, 366, 367, 369, 384, 394, 410, 414, 418, 422, 460, 467, 485, 491, 492, 498, 500, 533, 535, 540
hooks
OSDisableIntsHook() 159, 370, 382, 383
OSEnableIntsHook() 159, 370, 382, 383
OSIdlingHook() .. 158, 234, 235, 384, 385, 403, 484, 485, 497
monitor
OSRpt()165, 179, 187, 188, 189, 235, 239, 328, 329, 394, 495
other
OSIdle() 379
OSProtect() 372, 373
OSTimedOut() 169, 230, 243, 244, 366, 374, 375
OSUnprotect() 372, 373
OSVersion() 376, 377
tasks
OS_Delay() .. 4, 5, 26, 73, 75, 77, 96, 100, 135, 229, 235, 237, 241, 253, 254, 258, 259, 260, 261, 262, 263, 265, 268, 275, 291, 299, 311, 319, 323, 325, 335, 337, 340, 341, 345, 349, 379, 404, 464, 489, 497, 498, 505, 511, 512, 513, 536, 541
OS_DelayTS()..... 258, 260, 261, 310, 311, 340, 341, 354, 355
OS_Destroy() 262, 263, 298, 464, 535
OS_Prio() 238, 239, 334, 336, 464, 535
OS_Replace() 264, 265
OS_SetPrio() 77, 78, 266, 267, 300, 301, 302, 334
OS_Stop() 258, 261, 268, 269, 271, 352, 464, 535
OS_Yield() . 4, 63, 64, 65, 66, 67, 69, 70, 73, 75, 76, 100, 147, 167, 190, 232, 233, 236, 266, 267, 282, 283, 297, 341, 351, 359, 369, 379, 465, 486, 487, 488, 489, 498, 513, 532, 535, 541
OSCreateTask().... 4, 63, 64, 65, 66, 69, 73, 77, 100, 126, 167, 217, 235, 237, 238, 239, 244, 261, 262, 263, 264, 265, 266, 267, 269, 283, 296, 297, 298, 299, 330, 331, 337, 349, 350,

351, 353, 369, 381, 394, 402, 460, 466, 491, 498, 500, 533, 535, 541
OSDestroyTask() 264, 298, 299, 394
OSGetPrio() 140, 266, 300, 301, 302, 334, 394
OSGetPrioTask()..... 140, 300, 302, 303, 334, 336, 394, 406
OSGetState() 140, 304, 305, 306, 394
OSGetStateTask() 140, 304, 306, 307, 394, 406
OSGetTS() 261, 310, 311, 340, 341, 354, 394
OSSetEFlag() 141, 153, 273, 274, 275, 284, 320, 332, 333, 394
OSSetPrio() 77, 78, 238, 239, 266, 300, 302, 334, 335, 336, 394, 460
OSSetPrioTask() 300, 302, 334, 336, 337, 394
OSStartTask() 64, 126, 141, 217, 238, 268, 296, 297, 330, 350, 351, 352, 395, 406, 412, 416, 469, 495, 535, 536
OSStopTask()..... 268, 297, 352, 353, 395, 536
OSSyncTS() 261, 310, 340, 354, 355, 395
timer
OSGetTicks() 136, 231, 232, 308, 309, 338, 341, 394
OSSetTicks()..... 136, 231, 308, 338, 339, 394
OSSetTS()..... 261, 310, 340, 341, 354, 395
OSTimer()..... 74, 75, 100, 135, 136, 177, 194, 200, 208, 218, 228, 229, 231, 232, 238, 247, 250, 251, 258, 261, 356, 357, 395, 469, 486, 506, 507, 510, 531, 532, 533, 536, 539, 541

V

va_arg() 145
variables
accessing 107
declaring..... 387
errors when dereferencing..... 246
global, shared 67
initializing globals to zero..... 142
local..... 16, 19
locating in memory 89, 172–78
RAM required 458, 461
Salvo defined types 386
static 131, 236

W

Wagner, Thomas 519
watchdog timer..... 499

Y

Y2K compliance 215

Notes